Apeiron: A Scalable LLM-agentic Framework for Autonomous Full-lifecycle Demand-optimized Application Synthesis

Junyun Cheng[1*], Ankit Srivastava[2], Jessie Zeng[2], Milenko Drinic[2], Jack W. Stokes[2]

[1]Dartmouth College, [2]Microsoft

Correspondence: ankitsriv@microsoft.com

Abstract

We introduce Apeiron, a scalable and extensible framework for addressing *amorphous* user demands through autonomous, full-lifecycle application synthesis. Apeiron models the unstructured app development process as a heuristic optimization problem combining (i) a Computer-Use Agent (CUA) evaluator that simulates personas and demands, (ii) an *Activity Tracer* that grounds feedback in code-level interaction traces, and (iii) a *Locality Controller* that constrains changes during continuous integration and delivery (CI/CD). Furthermore, we introduce an innovative data generation approach using CUA-as-a-Judge to tackle data scarcity. Across 300 app scenarios, 2,400 personas, and 46,338 demands, Apeiron outperformed baselines by 10.7% in CUA ratings and 27.8% in user-demand task scores. The optimization process enhances task scores by 64.7%, and the tracer contributes a 25.1% gain. In CI/CD, Apeiron effectively restores 96.9% of the pre-shift mean CUA rating in one optimization step with <30% code changes in response to 30% demand shifts. Finally, a user study ($N = 18$) shows that our CUA ratings strongly correlate with human judgment (Spearman's $\rho = 0.685$) and that users prefer Apeiron-synthesized apps over baselines.

1 Introduction

Traditional, rigid, 'one-size-fits-all' apps are struggling in the contemporary landscape. Vast, volatile, and *evolving* user requirements are emerging (Dasanayake et al., 2019). This extensive, yet often ignored, *long-tail* niche requires attention (Ogrinz, 2009). Furthermore, there exist sensitive, secure, and *private* workflows involving confidential information, which are difficult to model. These *amorphous* demands motivate an autonomous, full-lifecycle synthesizer that can build and iteratively update apps within a domain. Rather than delivering a single fixed app, the system should adapt the app as demands shift over time. Although recent methods such as GPT-Engineer (Osika, 2023) and ChatDev (Qian et al., 2024) present software development capabilities, they operate mainly as single-use instruction-based systems that do not achieve the required full autonomy and demand awareness.

In this paper, we introduce Apeiron (Fig. 1), a framework that treats the app development process not merely as code generation, but as a **standardized program search problem** optimized against simulated user feedback. Unlike standard coding assistants that optimize for *correctness* (does it compile?), Apeiron optimizes for *utility* (does it satisfy the persona?). It achieves this via two key technical contributions: (1) A **Hybrid Feedback Loop** that couples high-level persona evaluations (Computer-use Agents) with low-level execution traces (Stracelit) to bridge the gap between user satisfaction and code logic; and (2) A **Locality Controller** that constrains the search space during Continuous Integration/Continuous Delivery (CI/CD) to ensure evolutionary rather than disruptive updates. While we demonstrate Apeiron within a Python Streamlit context, the underlying *search-and-optimize* paradigm offers a generalizable blueprint for autonomous software synthesis.

Inspired by LLM-as-Judge and Agent-as-a-Judge methods (Zheng et al., 2023; Chen et al., 2024a; Fu et al., 2024; Zhuge et al., 2025), we propose a CUA-as-a-Judge approach to evaluate the app quality. Its efficacy is demonstrated by a user study revealing a substantial correlation between CUA-based and human rankings (Spearman's $\rho = 0.735$). Additionally, we present a data synthesis strategy addressing data scarcity in this domain (Qian et al., 2024). Our experiments encompass 300 synthesized app specifications across 2,400 personas and 46,338 demands. We also develop an extended benchmark of 1,647

[*]Work done during an internship at Microsoft Research.

Findings of the Association for Computational Linguistics: ACL 2026, pages 3868–3899
July 2-7, 2026

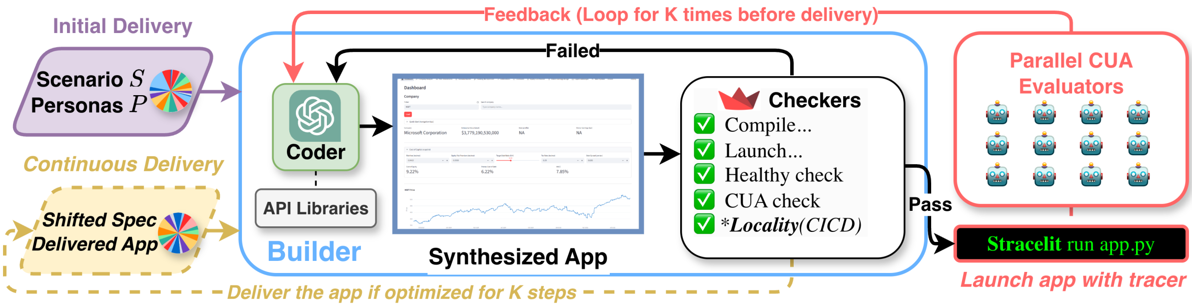


Figure 1: The Apeiron framework, illustrating the full-lifecycle synthesis process. A human user can optionally provide or dynamically update the target specifications. The Builder agent receives specifications for either Initial Delivery or Continuous Delivery and enters an iterative loop. It generates code, which is then verified by internal Checkers. If the checks pass, the app is launched with *Stracelit*, our Streamlit activity tracer, and evaluated by a parallel fleet of CUA agents. The resulting feedback is then routed back to the Builder to guide the next refinement cycle.

specifications and 255K demands. Results show that Apeiron produced apps outperform baselines by 10.7% in mean CUA rating and 27.8% in user-demand task score. Optimization boosts task scores by 64.7% while nearly halving the app error rate from 13.5% to 7.1%. Our hybrid feedback system yields a 15.5% CUA rating improvement. In simulated CI/CD scenarios, Apeiron effectively adapts to a 30% shift in user demands, recovering a 96.9% CUA rating after one optimization step. Our user study also shows a clear user preference for Apeiron-generated apps. To summarize, our contributions include:

- Introduce a demand optimized program search paradigm for full-lifecycle app synthesis.
- Build Apeiron, a scalable and extensible framework with hybrid CUA+tracer feedback and locality control for smooth CI/CD.
- Propose a CUA-as-a-Judge evaluation protocol and a data synthesis pipeline for demand-aware app benchmarks.

Our code and data are available at `https://github.com/microsoft/apeiron`.

2 Related Work

LLM-based Code Generation Early approaches emphasized graph representation (Zügner et al., 2021; Cheng et al., 2021a,b). Subsequently, foundational coding models (Chen et al., 2021; Li et al., 2022; Rozière et al., 2024) showed remarkable performance on algorithmic benchmarks (Chen et al., 2021; Austin et al., 2021). Later research examined training models with broader and more varied code datasets (Nijkamp et al., 2023; Xu et al., 2022; Li et al., 2023) and addressed challenges related to long-range dependencies in code (Wang et al., 2023). Recently, (FAIR CodeGen Team, 2025) introduced a world model for code generation, while (Romera-Paredes et al., 2024; Cheng et al., 2025; Novikov et al., 2025) expanded the scope of highly intricate scientific code. (Cheng et al., 2026) used a Jupyter Notebook coding process as a general reasoning process.

Multi-Agent Software Development The deployment of multi-agent systems has been suggested as a method to emulate software engineering teams with customized workflows (Qian et al., 2024; Osika, 2023; AI, 2024; OpenDevin, 2024). (Yang et al., 2024b) focused on dealing with code repositories. Studies by (Bionic-GPT, 2024) and (Packer et al., 2024) investigate long-term memory and the capacity to self-evolve. In addition, general-purpose agents (Wu et al., 2024; Chen et al., 2024b; Hong et al., 2024) have been utilized for software development purposes. However, most frameworks do not model demand-shift stages (CI/CD) or explicitly optimize over demands.

Automated Software Testing Automated testing is critical for app development. Early work focused on automated program repair (APR) (Xia et al., 2023; Ye et al., 2021; Le et al., 2022). More recent advances involve employing LLMs to generate unit tests (Schäfer et al., 2023), identify faults (Yang et al., 2024a), and perform automatic evaluations (Vu et al., 2024). Additionally, self-debugging and repair processes enhance the generated code by utilizing execution feedback (Chen et al., 2024c; Shinn et al., 2023; Gou et al., 2024; Xia and Zhang, 2023). Apeiron amalgamates these technologies while extending their application to user tests.

3 Full-lifecycle Application Synthesis

To achieve automation of the entire app lifecycle, we begin by abstracting the inherently unstructured software lifecycle in § 3.1. This enables us to formulate the synthesis process as a program search problem, as discussed in § 3.2.

3.1 A Conceptual Model of the App Lifecycle

Modern apps typically contain two phases: the *Initial Delivery*, responsible for developing a new app A from the ground up, and *CI/CD*, which involves the *incremental* modification of an existing app A to create an updated version A' based on new requirements. Both phases incorporate a three-stage development model: *requirement engineering* for generating a specification document, *build* for creating the app, and *evaluation* to offer feedback (Pressman, 2005). However, the CI/CD stage imposes a constraint that the updated app A' must remain behaviorally and interface-wise similar to A to avoid imposing a steep learning curve on users. We suggest a formal regulation to automate these procedures, which traditionally necessitated significant human involvement.

We standardize the app specification in a structured format, $Spec = (S, P)$, which serves as the primary input to our system. Here, S is a high-level *Scenario* that describes the app's overall purpose. For example, the description for a "Capital Allocation Insight Advisor" app scenario from our dataset:

> *Assists leadership in evaluating capital allocation trade-offs across organic growth, buybacks, and acquisitions. It draws on FMP owner earnings, enterprise values, and key financial ratios to compare internal projects...*

P represents a *Persona Distribution*, which is a weighted collection of user demands: $P = (\pi, L)$, where π is a description of the distribution, such as "Private Equity Portfolio Optimizers":

> *PE firms and their portfolio company leadership balancing deleveraging, add-on M&A, and organic growth. Speed, covenant awareness, and exit IRR drive decision-making, with strong emphasis on cash conversion...*

One persona distribution contains multiple specific personas with several different demands for each, defined over a list $L = \{(w_i, p_i, d_i)\}_{i=1}^{N_L}$ that includes N_L tuples of specific personas (p_i) and *demands* (d_i) represented as verifiable tasks with explicit expected outcomes and rubrics, and corresponding weights (w_i). For example, the "PE Operating Partner" persona:

> *Aged 40–60, ex-consultant or ex-operator, traveling between portfolio companies. Outcome-oriented and pragmatic, they prioritize EBITDA expansion, cash conversion, and time-to-value. They run weekly...*

and a demand "Portfolio-wide capital allocation stack-rank sprint" with 5% weight:

> *Aggregate portfolio projects, buybacks, and add-on M&A options; normalize cash flows via owner earnings; apply WACC...*
> ***Expected Outcome**: Produced a ranked portfolio capital allocation list with...*
> ***Rubric**: (1) Input owner earnings, EV, peer comps, covenants, FRED rates... (3) Output ranked tables, charts, and export...*

Detailed examples are presented in § E.2. Note that the same persona may have multiple demands. In the *Evaluation* stage, the app is tested against these demands (d_i) to generate feedback. The overall goal of app development is to deliver **demand-optimized** apps which best satisfy the demands of the list L. CI/CD is modeled as a *demand shifting* process, where the list L, which the original app A was optimized for, is *shifted* to a new list L' that the updated app A' needs to optimize, and the change is modeled as replacing $k\%$ of the items.

While we primarily model P as a broad distribution for diverse user bases, this formulation gracefully scales down to serve a single, specific human user. In a personalized deployment, the system can detect or be provided with a single persona carrying a 100% weight ($N_L = 1, w_1 = 1$). Furthermore, the user can dynamically update this specification at any time, treating their evolving needs as a continuous demand shift for CI/CD.

3.1.1 Conceptual-Model-Based Data Synthesis

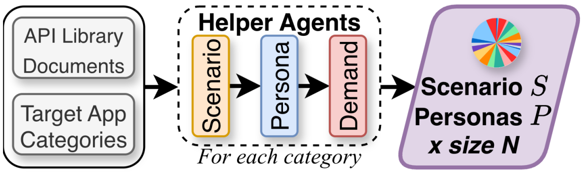


Figure 2: App lifecycle-based data synthesis pipeline.

In conjunction with this model, which structures the app specifications using a demand list-based evaluation, we also construct an LLM-based **data synthesis** pipeline. This pipeline is responsible for generating app specifications, which yield the dataset utilized in our experimentation. Illustrated in Fig. 2, the pipeline incorporates several *Helper Agents*. These agents receive a list of target app categories (such as technology, finance, and security) and API library documentation as inputs. They progressively synthesize unique scenarios, persona distributions, and demands incrementally for each category. Furthermore, these Helper Agents can also be employed as *requirement engineering assistants* in practice, aiding users in determining the specifications of their intended apps. We mitigate leakage via strict agent separation: The Builder synthesizes code purely from the high-level specification, without access to the demand-generation prompts. Full details are provided in § B.2.

3.2 App Development as Program Search

We formalize the development process as a program search problem to find the demand-optimized app that maximizes user utilities from an *App Domain* $\mathcal{D}$. This domain comprises the set of apps that can be constructed with reasonable effort (e.g., measured by lines of code) using available *API libraries*, since the functionalities of modern apps are largely derived from backend or cloud APIs they have access to. A **Program Synthesizer** ($B_D : (S, P, A_0, \gamma) \to A$), that integrates API libraries, samples apps from $\mathcal{D}$ given specs, with an optional initial app A_0, which may serve as a starting template or be used for CI/CD or refinement processes, and a feedback γ. In our framework, shown in Fig. 1, we design a Builder agent to play the role of synthesizer. An **Evaluator** ($R : (A, p_i, d_i) \to u$) offers utility evaluations u for a given app A and a specific demand d_i from persona p_i, implemented by CUAs in practice. The program searches for initial app delivery, and ongoing CI/CD phases are defined as follows.

Initial Delivery The goal is to find an app that maximizes the weighted sum of user utilities over all demands within the target persona distribution:

$$A_P^* = \underset{A \sim B_D(S,P,A_0,\gamma) \in \mathcal{D}}{\operatorname{argmax}} \sum_{i=1}^{N_L} w_i R(A, p_i, d_i) \quad (1)$$

where A_0 may represent either a project template originating from the framework (such as Streamlit in our experiments), an empty directory, or an existing app awaiting refinement. In practice, we approximate this objective via an agentic refinement framework that performs an evaluation-guided heuristic search. This process bridges the boundary between *ex-ante* forecasting and *ex-post* retrospective explanation. The specifications and personas act as *ex-ante* constraints. By autonomously executing the code and generating activity traces and rubrics, Apeiron pulls retrospective user-experience signals forward, transforming them into an optimization signal before the application is delivered to the user. The optimization step corresponds to the Builder generating a new code mutation A_{t+1} conditional on γ_t, effectively exploring the discrete space of code modifications in a gradient-free manner.

Continuous Delivery with Locality Constraints As elaborated above in § 3.1, CI/CD introduces a *locality* constraint necessitating a smooth transition by reducing disruptive alterations to the app:

$$A_{P'}^* = \underset{A' \sim B(S,P',A_P^*,\gamma) \in \mathcal{D}}{\operatorname{argmax}} \sum_{j=1}^{N_L'} w_j' R(A', p_j', d_j') \quad (2)$$

$$\text{subject to } \operatorname{diff}(A', A_P^*) < \delta \quad (3)$$

where an updated app $A_{P'}^*$ optimizes user satisfaction for the shifted needs P' based on a prior app A_P^*, $\operatorname{diff}(A', A_P^*)$ quantifying the variations between the apps, and $\delta \in [0, 1]$ governs the transition smoothness. In our experiments, we apply a basic *code turnover rate* based on the total number of lines of code (LoC) added and removed, normalized by the initial line count, while advanced, semantic-based methods are also possible.

4 The Apeiron Framework

Apeiron instantiates the program search in § 3 as a build–evaluate–refine loop (Fig. 1). A Builder

Agent serves the synthesizer that writes code (§ 4.1), and a multi-CUA Evaluator provides feedback (§ 4.2). They operate iteratively, optimizing the app around user demands (§ 4.3). Apeiron also allows large-scale app synthesis and is easily extensible to new app domains and frameworks (§ 4.4).

4.1 Builder Agent

The Builder Agent operates in an iterative, multi-step process within a sandboxed environment:

1. **Code Generation**: The agent receives $Spec = (S, P)$ and any feedback, then outputs a sequence of file operations (e.g., write/delete 'app.py') on the app codebase.

2. **Internal Checks**: Before submitting the app to the CUA evaluator, the app needs to pass internal checks inspired by pair programming (Hannay et al., 2009) to ensure the app can be launched successfully and healthily, as well as having basic functionality, via rule-based checkers and a single fast CUA checker. The CUA checker shares the same structure as the CUA Evaluators introduced later; however, it is prompted to traverse the app rapidly to troubleshoot functionality before launching expensive parallel CUA evaluations. If any checks fail, the Builder must resolve them.

3. **Submission**: Once the internal checks pass, the Builder can either submit the app to the CUA Evaluators or do more refinements.

The prompts can be found in § F.1. We perform experiments in Streamlit sandboxes as detailed in § B.4 using the business API libraries in § B.3.

4.2 CUA Evaluators

Prior work explores using LLMs as proxies for human evaluation (Gu et al., 2025). In our study, we utilize a CUA to assess the quality of synthesized apps using OpenAI's computer use models (OpenAI, 2025). It iteratively outputs browser operations based on the latest screenshot. We cap each CUA session at 30 interaction steps (mouse/keyboard actions). The CUA's prompt can be found in § F.2. The evaluation is performed as follows:

1. **Task Sampling**: A batch of persona-demand pairs (p_i, d_i) is sampled from the target list L.

2. **Parallel Evaluation**: For each pair, a CUA instance is instantiated with p_i and tasked with completing the demand d_i by interacting with a running instance of A.

3. **Feedback Generation**: Upon completion, each CUA generates a structured report containing multi-dimensional ratings (Functionality, Usability, etc.) about user experience (ISO, 2019) from 1 to 10, a qualitative review, and a final outcome o of success, failure, or error. Examples can be found in § E.1.

Algorithm 1 Apeiron Optimization Loop

Input: Spec S, P, Max Steps K, Threshold δ
Output: Optimized App A^*
$A_0 \leftarrow$ Builder$(S, P, \emptyset)$ ▷ Initial Build
for $t = 1$ **to** K **do**
    Pass $\leftarrow$ Checkers(A_{t-1})
    if $\neg$ Pass $\vee$ *Need more refinement* **then**
        $A_{t-1} \leftarrow$ Fix(A_{t-1}); **continue**
    end if
    $Traces \leftarrow$ Stracelit(A_{t-1})
    $Feedback \leftarrow$ ParallelCUA(A_{t-1}, P)
    if CI/CD **and** Diff$(A_{t-1}, A_{new}) > \delta$ **then**
        Reject A_{new}; **continue** ▷ Locality
    end if
    $A_t \leftarrow$ Builder$(A_{t-1}, Feedback, Traces)$
end for
return A_K

We use the same CUA configuration as an automated judge to score apps. We use a trifold way to ensure reliability of CUA judges: (1) **Objective Metrics**: Error Rate and Unfinished Rate are binary runtime states independent of LLM judgment; (2) **Human Validation**: As detailed in § C, our CUA ratings highly align with human annotators ($\rho = 0.735$); and (3) **Ablation Consistency**: The degradation of performance when removing CUAs (Fig 8) indicates information gains.

4.3 Feedback-Driven Optimization Loop

As shown in Alg. 1, Apeiron performs multiple optimization steps after the initial generation; each step involves a CUA evaluation and subsequent refinement by the builder. It integrates an activity tracer, which transforms a high-level user experience into practical, code-level insights, along with a locality controller for smooth CI/CD.

Streamlit Activity Tracer CUA reports deliver qualitative insights without a clear causative relationship between user behaviors and app functionalities that lead to the given experience. We introduce a Streamlit activity tracer, "Stracelit", which intercepts an active app and produces two artifacts:

- **Abstract Component Tree (ACT)**: A hierarchical representation of the app's structure, where each node corresponds to a specific widget and its source file and code location.

- **Action Traces**: A chronological log of every user action and subsequent app response, with each event mapped to an ACT node.

Examples can be found in § E.1. The high-level CUA report and the low-level traces form a comprehensive blueprint of the app's structure, connecting the resulting user outcome and the code that caused it. Further details are provided in § B.5.

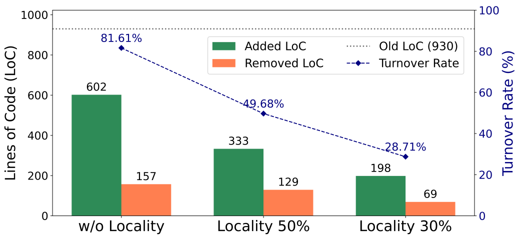


Figure 3: Effect of locality controller on code turnover.

Locality Controller The Locality Controller functions as part of the internal checkers, employing *Git diff* to calculate the text-based code turnover rate (Lines of Code added/removed). Should this turnover rate surpass a predefined threshold δ, the suggested modifications are declined. We found that structural alternatives, such as calculating the Tree Edit Distance (TED) on Abstract Component Trees, were highly brittle to benign layout refactoring (e.g., wrapping existing UI elements inside a new container drastically shifts the tree depth). Furthermore, exact AST edit distances mathematically converge with simple statement turnover. Thus, text-based code turnover serves as a highly efficient proxy for semantic difference. As depicted in Fig. 3, in the absence of locality control, the agent tends to extensively modify the code with $\sim$80% turnovers.

4.4 Scalability and Extensibility

Allowed by our highly decoupled design, where we ensure each building step and CUA instance are independent and isolated, Apeiron achieves massive parallelization, allowing high-throughput app synthesis. As shown in Fig. 12 (§ B.7), the system uses a workload queue to distribute specifications *Spec* that contain scenario-persona pairs (S, P) to a pool of independent Builder workers. When a Builder submits an app, it is dispatched to another pool of CUA workers. This decoupled, many-to-many architecture allows a large number of CUA tests to be conducted in parallel. Apeiron is also highly extensible along two primary axes:

- **Domain Extension**: As introduced in § 3.1, the app domain is defined by the API libraries provided, thus Apeiron adapts to new domains simply by linking to the new libraries.

- **Framework Extension**: Apeiron can be adapted to other frameworks (e.g., Reflex, Taipy) by implementing a new 'Compiler' interface for the program runtime. For desktop applications, this involves compiling the source code and launching it in a headless mode (e.g., via Electron in a Chromium container) to return an interface for control signals. For server-side backend software, the CUA can simply be substituted with a standard API-testing LLM agent.

5 Experiments

We conduct a comprehensive set of experiments. The setups are introduced in § 5.1. We focus on three primary research questions:

- **RQ1:** How effective is Apeiron's feedback-driven optimization at improving app quality compared to unoptimized generation and state-of-the-art baselines? (§ 5.2)

- **RQ2:** What are the distinct contributions of the different feedback signals (high-level CUA reports vs. low-level activity traces) to the optimization process? (§ 5.3)

- **RQ3:** How effectively does Apeiron adapt to shifting user demands in CI/CD, and what is the impact of the locality controller? (§ 5.4)

Further ablation studies are presented in § D. Statistical significance of our results is studied in § D.1.

5.1 Experimental Setup

Dataset We create a large-scale LLM synthesized dataset using the method in § 3.1.1, which yields 300 unique app specifications across six categories (business, economics, finance, government, technology, and marketing), 2400 distinct personas

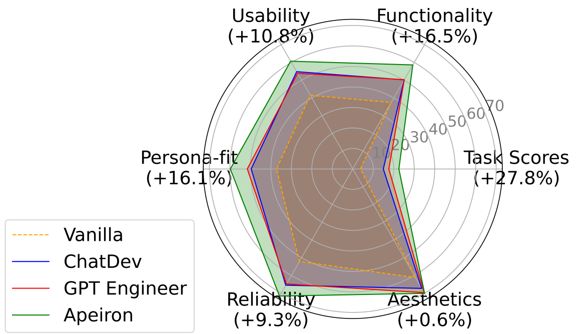


Figure 4: Comparison of Apeiron (Opt. Step 3) against baselines on CUA Ratings.

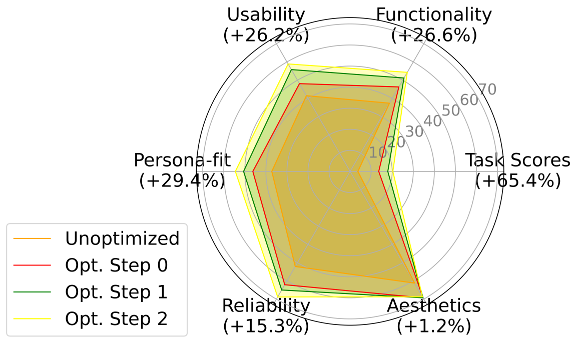


Figure 6: CUA ratings across three optimization steps and the initial unoptimized build.

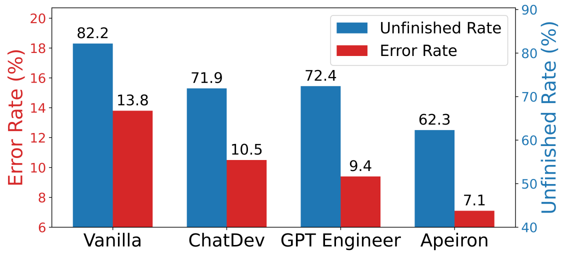


Figure 5: Error and unfinished rate against baselines.

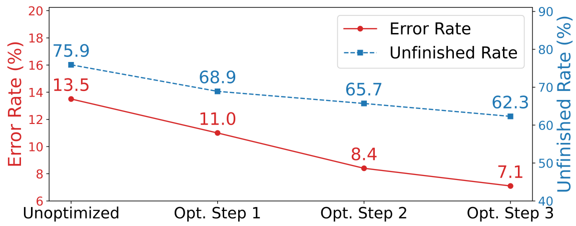


Figure 7: Error and unfinished rates by optimizing steps.

with 46,338 user demands. We further develop an extended benchmark with 1647 specs and 255K demands introduced in § B.6.

Baselines We compare our method against three representative baselines to isolate the impact of our optimization loop. **Vanilla** represents direct LLM synthesis. **ChatDev** (Qian et al., 2024) and **GPT-Engineer** (Osika, 2023) represent state-of-the-art and widely used *scaffolded* generation paradigms. Note that these baselines typically operate as one-shot or waterfall systems. To compare them with Apeiron's iterative approach, we align the evaluation metrics while acknowledging potential differences in compute budgets. This comparison is intended to demonstrate the specific value of our proposed *iterative optimization paradigm* versus the standard 'fire-and-forget' generation models.

Implementation Details Our method and baselines are based on gpt-5-2025-08-07; we perform an ablation study on the base model selection in § D.3. For each app, the initial build is followed by 3 optimization steps, with a batch size of 20 demands used for CUA evaluation at each step. The ablation over batch size is provided in § D.2 and detailed hyperparameters can be found in § B.1.

Evaluation Metrics We use a hybrid of CUA-report and objective metrics to assess app quality:

- **CUA Ratings**: Average user-centric scores (1-10) across five dimensions from the CUA evaluator discussed in § 4.2: *Functionality*, *Usability*, *Persona-fit*, *Reliability*, and *UI/Aesthetics*. Details can be found in § F.2.
- **Task Score**: A weighted sum of success over all demands $\sum_i w_i \mathbb{1}_{[o_i = success]}$.
- **Error Rate (%)**: The percentage of CUA sessions that terminated due to a critical app error (e.g., a crash or bug).
- **Unfinished Rate (%)**: Ratio of tasks CUA was unable to conclude within 30 interaction steps, excluding error sessions. Measuring if tasks can be done *efficiently* within limited steps with the given app.

5.2 RQ1: Impact of Optimizations

First, we compare Apeiron against SOTA baselines. As shown in Fig. 4, Apeiron achieves superior performance across all CUA rating dimensions, with a mean rating improvement of 10.7% over the second-best model. Fig. 5 further shows that Apeiron is more efficient and robust, reducing the Unfinished Rate by 14.0% and the Error Rate by 24.5% compared to GPT-Engineer.

Next, we analyze the performance gain over optimization steps. Fig. 6 shows that CUA ratings exhibit stable improvement with each refinement

	Code-level	Qualitative	Rating
Full	✓	✓	✓
w/o Tracer		✓	✓
w/o CUA			✓

Table 1: Ablation study settings.

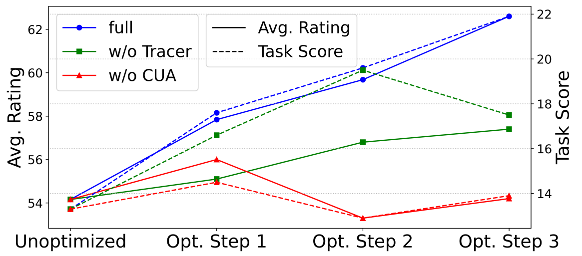


Figure 8: Average rating and task score across optimization steps for the full and ablated setups.

cycle, most notably in Persona-fit (+29.4%), Usability (+26.2%), and Functionality (+26.6%). The overall Task Score sees the largest gain, increasing by 65.4% from the initial build. Fig. 7 further shows that the Error Rate is nearly halved, decreasing from 13.5% to 7.1%, while the Unfinished Rate drops steadily from 75.9% to 62.3%. It suggests that Apeiron has benefited from its iterative, feedback-driven optimization process.

5.3 RQ2: Impact of Feedback Modalities

We conducted an ablation study to isolate the contributions of Apeiron's different feedback components. We compared the full system against two variants: one without our low-level "Stracelit" activity traces (**w/o Tracer**), and one removing CUA reports with only numerical ratings left (**w/o CUA**), as detailed in Table 1.

The results in Fig. 8 highlight the critical role of both feedback types. The performance of the **w/o CUA** setting collapses, indicating that scaler ratings alone are insufficient for meaningful optimization; the Builder requires the qualitative report

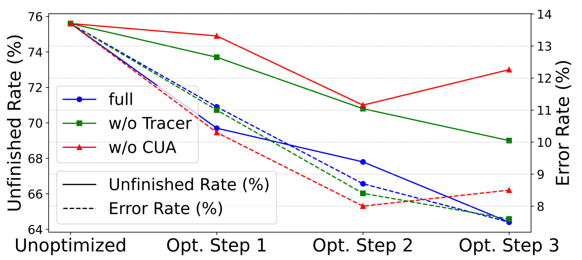


Figure 9: Error/unfinished rates in feedback ablation.

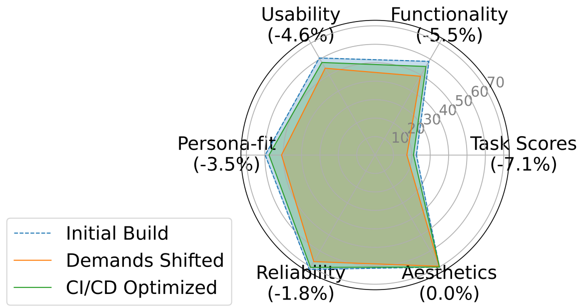


Figure 10: CUA Ratings for an app after its initial build, after a 30% demand shift, and after CI/CD optimization.

	Free	50% LC	30% LC	10% LC
Score	19.4	19.5 (0.5%)	18.5 (4.6%)	17.1 (11.9%)
Rating	61.2	61.0 (0.3%)	59.0 (3.6%)	57.6 (5.9%)
Error	8.6	8.7 (1.2%)	9.1 (5.8%)	9.9 (15.1%)
Unfin.	68.2	70.0 (2.6%)	73.0 (7.0%)	74.8 (9.7%)

Table 2: Impact of different Locality Control (LC) thresholds during the CI/CD phase, with percent change from the unconstrained "Free" setting. Blue/red indicates improvement/degradation.

to understand *why* a feature is failing. The **w/o Tracer** setting shows some improvement, but is consistently outperformed by the **Full** system. This shows that while high-level feedback provides direction, code-based signals from Stracelit are essential to enable precise code modifications. Specifically, tracer improves average rating and task score by 9.1% and 25.1% , respectively, while the CUA provides 15.5% and 57.6% improvements. This trend is mirrored in the robustness and efficiency of the app, as shown in Fig. 9, further confirming the importance of high- and low-level feedback.

5.4 RQ3: Impact of Locality Constraints

To simulate a real-world CI/CD scenario, we took fully optimized apps and "shifted" their demand list L by replacing 30% of demands with new ones. As shown in Fig. 10, this shift causes a significant drop of 23.1% and 9.4% in task score and avg. rating respectively. However, after a single CI/CD optimization step, Apeiron is able to recover the pre-shift user experience levels with 92.9% in Task Scores and 96.9% avg. rating.

We also evaluated the impact of different Locality Control thresholds. Table 2 shows a clear trade-off: stricter controls (e.g., 10% LC) lead to a higher degradation in metrics as the Builder has less freedom to adapt the app while a loose control allows better recovery with a risk of significant modifications to the app.

6 Discussion

While Apeiron presents a blueprint for autonomous application synthesis, several future directions can move us closer to this vision. First, implementing *lifelong learning* capabilities is crucial for the system to continuously evolve its competencies. This enables enhanced *personalization*, where a human-in-the-loop can interact with the system to explicitly update their individual persona spec over time. By progressively aligning the CUA's evaluation rubric with the actual end-user's live feedback, the framework can adapt apps to highly specific, idiosyncratic workflows. Second, our current locality controller relies on structural code turnover. Future iterations should incorporate higher-level invariants, such as visual UI graph matching to prevent unexpected layout shifts, or trace-level assertions (running previous Activity Traces as regression tests) to ensure prior workflows remain functional despite new code. Finally, while Apeiron successfully automates the classical software engineering lifecycle, modern DevOps encompasses a much richer ecosystem. Extending this framework to model continuous experimentation, observability-driven iteration, and socio-technical feedback loops remains an exciting frontier for future work.

7 Conclusion

In this paper, we present Apeiron, a system for autonomously synthesizing full-lifecycle apps that optimizes for the amorphous demands of users. We formalize the inherently unstructured app lifecycle as a program search problem. This model underlies our scalable and extensible Apeiron framework, enabling a CUA-as-a-Judge evaluation and a data synthesis method overcoming data scarcity. We also introduce a productive low-level activity tracer designed to enhance CUA feedback and a locality controller aimed at facilitating smooth CI/CD. Experimental results demonstrate that our integrated methodology yields robust, high-quality apps that notably surpass established baselines, pointing toward a future in which interactive applications can be synthesized and evolved on demand.

Limitations

The performance of Apeiron is fundamentally dependent on the quality and comprehensiveness of its bound API library. A sparse or poorly documented library will naturally limit the complexity and functionality of the apps that can be synthesized. Additionally, while our CUA evaluation shows strong correlation with human judgment (§ C), it remains a simulation and may not capture all subtleties of human-computer interaction. Also, because the optimization loop uses CUA feedback, a remaining risk is overfitting to the automated evaluator; cross-judge evaluation is an important direction for future work. Finally, the iterative optimization introduces higher token and time costs compared to one-shot baselines. We position Apeiron as a solution for high-value, custom workflows where this compute overhead is justified by the significant gains in utility and reliability compared to unoptimized generation.

Ethical considerations

As a foundational research for autonomous application synthesis, we do not observe any apparent ethical concerns or risks.

Use of LLMs statement

We primarily use LLMs to polish the writing and check for typos.

References

Cognition AI. 2024. Devin. https://cognition.ai/blog/introducing-devin.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Bionic-GPT. 2024. Bionic-gpt. https://github.com/bionic-gpt/bionic-gpt.

Dongping Chen, Ruoxi Chen, Shilin Zhang, Yaochen Wang, Yinuo Liu, Huichi Zhou, Qihui Zhang, Yao Wan, Pan Zhou, and Lichao Sun. 2024a. Mllm-as-a-judge: assessing multimodal llm-as-a-judge with vision-language benchmark. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 34 others. 2021. Evaluating large language models trained on code.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024b. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024c. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Junyan Cheng, Peter Clark, and Kyle Richardson. 2025. Language modeling by language models. *Preprint*, arXiv:2506.20249.

Junyan Cheng, Iordanis Fostiropoulos, and Barry Boehm. 2021a. Gn-transformer: Fusing sequence and graph representation for improved code summarization. *Preprint*, arXiv:2111.08874.

Junyan Cheng, Iordanis Fostiropoulos, and Barry Boehm. 2021b. Graph conditioned sparse-attention for improved source code understanding. *Preprint*, arXiv:2112.00663.

Junyan Cheng, Kyle Richardson, and Peter Chin. 2026. Analytica: Soft propositional reasoning for robust and scalable LLM-driven analysis. In *The Fourteenth International Conference on Learning Representations*.

Sandun Dasanayake, Sanja Aaramaa, Jouni Markkula, and Markku Oivo. 2019. Impact of requirements volatility on software architecture: How do software teams keep up with ever-changing requirements? *Journal of Software: Evolution and Process*, 31(6).

Meta FAIR CodeGen Team. 2025. Cwm: An open-weights llm for research on code generation with world models.

Jinlan Fu, See-Kiong Ng, Zhengbao Jiang, and Pengfei Liu. 2024. GPTScore: Evaluate as you desire. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6556–6576, Mexico City, Mexico. Association for Computational Linguistics.

Zhibin Gou, Zhihong Shao, Yeyun Gong, yelong shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2024. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations*.

Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Yuanzhuo Wang, Wen Gao, Lionel Ni, and Jian Guo. 2025. A survey on llm-as-a-judge. *Preprint*, arXiv:2411.15594.

Jo E Hannay, Tore Dybå, Erik Arisholm, and Dag IK Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information and software technology*, 51(7):1110–1122.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

ISO. 2019. Ergonomics of human-system interaction – part 210: Human-centred design for interactive systems. Standard ISO 9241-210:2019, International Organization for Standardization, Geneva, CH.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. 2022. Coderl: mastering code generation through pretrained models and deep reinforcement learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, and 39 others. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *Preprint*, arXiv:2506.13131.

Michael Ogrinz. 2009. *Mashup patterns: Designs and examples for the modern enterprise*. Pearson Education.

OpenAI. 2025. Computer-using agent. (`https://openai.com/index/computer-using-agent/`).

OpenDevin. 2024. Opendevin. `https://github.com/OpenDevin/OpenDevin`.

Anton Osika. 2023. Gpt engineer. `https://github.com/gpt-engineer-org/gpt-engineer`.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Memgpt: Towards llms as operating systems. *Preprint*, arXiv:2310.08560.

Roger S Pressman. 2005. *Software engineering: a practitioner's approach*. Palgrave macmillan.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, Bangkok, Thailand. Association for Computational Linguistics.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2024. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *Preprint*, arXiv:2302.06527.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Tu Vu, Kalpesh Krishna, Salaheddin Alzubi, Chris Tar, Manaal Faruqui, and Yun-Hsuan Sung. 2024. Foundational autoraters: Taming large language models for better automatic evaluation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17086–17105, Miami, Florida, USA. Association for Computational Linguistics.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088, Singapore. Association for Computational Linguistics.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494. IEEE Press.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *Preprint*, arXiv:2301.13246.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA. Association for Computing Machinery.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024a. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024b. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2).

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2025. Agent-as-a-judge: Evaluate agents with agents. In *Forty-second International Conference on Machine Learning*.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*.

A The "Amorphware" Paradigm

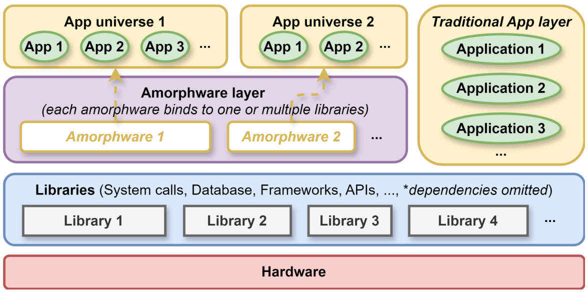


Figure 11: The conceptual model of the Amorphware paradigm. **Top Right:** The traditional application layer, where users must select from a finite set of rigid, prefabricated applications optimized for broad, static demands. **Top Left:** The proposed Amorphware layer, where a generative synthesizer mediates between raw API libraries and user intent. This layer dynamically instantiates ephemeral software tailored to the specific, momentary needs of the user, effectively creating a unique "App Universe" for every in-domain demand.

In this section, we discuss the broader architectural implications of Apeiron. The limitations of the current software landscape lie in its rigidity; users are forced to collect a vast suite of "one-size-fits-all" applications, attempting to patch together workflows that rarely fit their unique requirements perfectly. This results in a friction-heavy experience where the user must adapt to the software, rather than the inverse.

We propose the **Amorphware Paradigm** to address this fundamental misalignment. As illustrated in Fig. 11, this paradigm introduces a generative intermediary—the *Amorphware Layer*—situated between the user and the foundational digital infrastructure (APIs, databases, and system calls). Unlike the traditional application layer, which delivers fixed artifacts, the Amorphware layer delivers *capabilities*. It functions as a just-in-time synthesizer that accepts amorphous, unstructured user demands and instantiates bespoke software interfaces on the fly.

This suggests a computing future where the "App Store" is replaced by an "Ability Store"—a repository of domain-specific synthesizers rather than static executables. In this future, software is no longer a fixed product but a fluid, adaptive service that evolves continuously alongside the user's intent. Apeiron represents a concrete, first-step realization of this layer, demonstrating that high-utility, demand-optimized applications can indeed be synthesized from the ground up.

B System and Experiment Details

B.1 Hyperparameters

General		CUA	
exception_retries	3	max_iterations	30
max_func_calls	5	max_opt_steps	3
max_llm_recall	3	global_cua_lock	300
random_seed	42	batch_size	20
Builder		**CI/CD**	
max_build_steps	5	locality_ratio	0.3
max_session_ops	25	n_demands_shift	20

Table 3: Hyperparameters in our experiments.

The hyperparameters used in our experiments are listed in Table 3 that control the agents' operation, evaluation, and optimization loops.

General These parameters define the foundational behavior of the LLM agents.

- `exception_retries`: The maximum number of times an agent will retry an operation after an internal error or exception before failing.
- `max_func_calls`: The maximum number of consecutive tool calls (e.g., file I/O, API calls) an agent can make in a single turn before it must generate a natural language response.
- `max_llm_recall`: The number of times an agent can re-prompt the underlying LLM with the full conversation history in case of repeated parsing errors or failures, forcing a re-evaluation of its strategy.
- `random_seed`: A seed used for all stochastic processes, such as task sampling and data splitting, to ensure the reproducibility of our experiments.

CUA (Computer Use Agent) These parameters control the automated evaluation process.

- `max_iterations`: The maximum number of actions (e.g., clicks, key presses) a CUA is allowed to perform when attempting to complete a user demand. This prevents infinite loops and ensures timely evaluation.

- `max_opt_steps`: The total number of iterative refinement cycles (Build → Evaluate → Feedback) performed for each application during the initial delivery phase.

- `global_cua_lock`: The maximum number of CUA instances that can run concurrently across the entire system, managed by a semaphore to prevent resource exhaustion.

- `batch_size`: The number of user demands randomly sampled from the available pool to be evaluated by the CUA workers in each optimization step.

Builder These parameters govern the code generation agent's workflow.

- `max_build_steps`: Within a single optimization cycle, this is the maximum number of attempts the Builder agent can make to fix compilation or runtime errors before the cycle is considered a failure.

- `max_session_ops`: The maximum number of conversational turns or file system operations (e.g., writing multiple files) the Builder can perform before it is required to submit the code for compilation and checking.

CI/CD (Continuous Integration and Delivery These parameters are specific to the continuous delivery stage, where the application is updated to meet new demands.

- `locality_ratio`: The maximum allowed code turnover rate (defined as the sum of added and deleted lines of code divided by the original line count) for an update. This enforces that changes are localized and incremental, preventing the agent from rewriting the entire application.

- `n_demands_shift`: The number of existing user demands that are replaced with new ones to simulate a shift in user requirements during the CI/CD phase.

B.2 Dataset Details

To generate a diverse and realistic set of development tasks and due to the scarcity of data in this field, we synthesize the dataset by a dedicated agent; the prompts can be found in § F.3. This ensures that each task is grounded in the capabilities of the system's library and reflects a plausible user need. The statistics of the dataset used in our experiment are shown in Table 4, with each scenario we consider 8 different distributions, while each distribution contains around 70 distinct user demands.

	Scenarios	Personas	Demands
Business	50	400	7669
Technology	50	400	7682
Economics	50	400	7759
Government	50	400	7719
Finance	50	400	7854
Marketing	50	400	7655
Total	300	2400	46338

Table 4: Dataset statistics.

Stage 1: Scenario Synthesis The process begins with a `scenario_helper` agent that generates high-level application **Scenarios**. By analyzing the available API library (e.g., financial and economic data APIs), the agent proposes relevant application concepts within a given domain. Each scenario consists of a name, a detailed description of the application's purpose, and its category.

Stage 2: Persona Synthesis For each generated scenario, a `persona_helper` agent creates multiple distinct `PersonaDistributions`. A distribution is a set of user **Personas**, each with a detailed description of their professional background, technical skills, goals, and their proportional representation within the target user base. This stage models the varied user groups an application might serve. Note that in a live deployment, rather than synthesizing hypothetical users, these helper agents can act as interactive profilers (e.g., via a conversational interface) to detect and formalize a real human user's specific persona and update their active distribution.

Stage 3: Demand Synthesis Finally, a `demand_helper` agent takes a scenario and a persona distribution to generate fine-grained user **Demands**. For each persona, it synthesizes specific tasks they would perform with the application. Crucially, each demand includes not only a description but also a clear `expected_outcome` and a detailed evaluation `rubric`, which are later used by the CUA evaluators to programmatically assess task completion and application quality.

B.3 API Library

ID	Description	#
`fmp`	**Financial Modeling Prep API:** *FMP provides the Stock Market APIs and Financial Data APIs, such as real-time stock prices, financial statements, and historical data. It offers a comprehensive solution to meet all financial data needs.*	132
`fred`	**Federal Reserve Economic Data API:** *The FRED® API retrieves economic data from the FRED® and ALFRED® websites hosted by the Economic Research Division of the Federal Reserve Bank of St. Louis.*	16

Table 5: The library of external data APIs available to the agent. Each proxy provides access to a suite of specific endpoints. The '#' column indicates the number of endpoints for each proxy.

The capabilities of the Builder agent, and thus the domain of synthesizable applications, are defined by an extensible library of external APIs. In our experiments, this library provides access to financial and economic data through two primary proxy services: Financial Modeling Prep (FMP) and Federal Reserve Economic Data (FRED), which are useful in building business-focused applications in our experiments.

All API interactions are managed through a modular proxy system. This design allows for easy integration of new data sources by implementing a new subclass. The Builder agent interacts with this system via a single, sandboxed function, `CALL_API`, which is dynamically injected into its environment during the build process.

To ensure correct usage, the agent is instructed to first query for API specifications using a dedicated `retrieve_api_doc` function before making a call. Table 5 details the APIs available to the agent.

B.4 Streamlit Environment

To ensure safe and isolated application builds, Apeiron executes all code within a sandboxed environment managed by the `StreamlitCompiler` class. This sandbox is critical for isolating dependencies and enabling the parallelized testing required for large-scale synthesis.

For each build session, a temporary, isolated virtual environment is created using Python's native `venv` module. This practice prevents dependency conflicts between different application builds and ensures that each app is self-contained. Dependencies specified in the application's `requirements.txt` file are installed asynchronously into this sandboxed environment before execution. Moreover, it will install other tools like the tracers.

The application is then launched as a separate, non-blocking subprocess using `asyncio`. Crucially, it is run with the `-server.headless true` flag, allowing the app to run without a graphical user interface, which is essential for automated evaluation. All standard output and error streams are redirected to a log file, enabling the framework's `Checker` module to programmatically detect compilation errors, monitor runtime health, and confirm that the application has launched successfully before handing off to the CUA evaluators.

B.5 Streamlit Tracer

While CUA feedback provides essential high-level, qualitative insights, it often lacks the code-level precision required for an autonomous builder to perform targeted fixes. To bridge this gap, we developed `Stracelit`, a custom activity tracer that instruments the Streamlit framework to generate low-level feedback.

The tracer is activated by wrapping the application's execution command; instead of `streamlit run`, the system invokes `stracelit run`. This wrapper injects our tracing logic by monkey-patching the `streamlit` module at runtime. All calls to Streamlit's API are intercepted by a `TracerWrapper` class.

For each intercepted call (e.g., `st.button`), the wrapper inspects the call stack to identify the precise file and line number in the application's source code where the call originated. To provide richer context, a `SourceCache` utility parses the application's source files into Abstract Syntax Trees (ASTs), allowing the tracer to map a single line number to the full multi-line statement or context manager it belongs to.

This process generates two key artifacts:

1. **Abstract Component Tree (ACT):** A hierar-

chical representation of the application's UI structure. Each node in the tree corresponds to a specific widget or layout element and is directly linked to the source code that generated it.

2. **Action Traces:** A chronological log of events that connects user interactions (e.g., a button click) to the specific component in the ACT that received the event, as well as any subsequent application responses.

Together, these artifacts provide the Builder agent with a detailed record of the application's structure and dynamic behavior, enabling precise, targeted code modifications during the optimization cycles. We provide examples for the ACT, traces, as well as CUA reports in § E.1.

B.6 Extended "Amorphware" Benchmark

	Scenarios	PersonaDist.	Demands
Busi.	50	275	42630
Tech.	56	293	45150
Econ.	50	268	41586
Gov.	50	267	41071
Fin.	50	265	41433
Market.	50	279	43140
Total	306	1647	255010

Table 6: Amorphware Benchmark.

Notice that each PersonaDistribution contains 8 different personas. We pair a scenario and a persona distribution as a specification.

B.7 Apeiron Parallel Synthesis

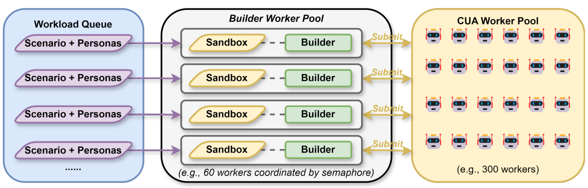


Figure 12: Apeiron's decoupled, many-to-many architecture for high-throughput synthesis and evaluations.

C CUA Consistency with Human Judgment

To validate our automated evaluation framework, we conducted an expanded user study to measure the consistency between the CUA's assessments and those of human evaluators, ensuring our framework optimizes for genuine user utility rather than evaluator-specific biases. The study was designed to answer two questions: (1) Do the CUA's rankings of application quality correlate with human rankings across diverse professional domains? (2) Do human users prefer applications generated by Apeiron over those from strong baseline systems?

C.1 Experimental Setup

Participants and Stimuli To ensure robust and diverse feedback, we scaled our participant pool to 18 volunteers spanning various professional backgrounds, technical proficiencies, and age groups, purposefully mirroring the target personas of our commercial application scenarios. The detailed participant demographics are provided in Table 7.

Demographic Category	Distribution (Count)
Academic/Professional Background	Business & Management (6), Natural & Biomedical Sciences (5) Engineering & CS (7),
Working Experience	0 years (6), 0–3 years (8), 3–5 years (3), 5+ years (1)
Age Group	18–22 (5), 22–28 (7), 28–35 (5), 35+ (1)

Table 7: Demographics of the 18 participants in the human evaluation study.

The stimuli consisted of 30 applications, randomly selected from our experiment in § 5.2, 10 from each method (Apeiron, GPT Engineer, and ChatDev). Each participant was provided with the app, the scenario, persona distribution, and demands, as well as the same instructions we provide to the CUA as a reference to provide the rating, using the same 5-dimensional score. We take the average rating to obtain the ranking of the apps. The materials were provided to the participants as a PDF.

Procedure The study followed a two-part procedure:

1. **Ranking Task:** To ensure robust data, each of the 30 applications was evaluated by 3 different volunteers. Each volunteer was assigned 15 applications to evaluate. For each application, they were provided with the exact same context as a CUA agent in a document as a reference: the high-level scenario, the specific user persona they were to embody, and a list of user demands. Participants were asked to

interact with the application to complete the demands and then rate its overall quality based on the same criteria used by the CUA (Functionality, Usability, etc.). From these ratings, we calculated a mean human rank for each of the 30 applications (where a rank of 1 is best).

2. **Pairwise Preference Task:** After the ranking task, participants were presented with pairs of applications—one from Apeiron and one from a baseline—and asked to indicate which one they preferred for accomplishing the specified tasks. This resulted in a direct head-to-head comparison of user experience.

C.2 Results

Rank Correlation and Subgroup Analysis We find a strong, statistically significant global correlation between the automated CUA rankings and the mean human rankings. As shown in Fig. 13, we calculated a Spearman's rank correlation coefficient of $\rho = 0.685$ ($p < 0.001$). This result indicates that the CUA is a valid and reliable proxy for human judgment.

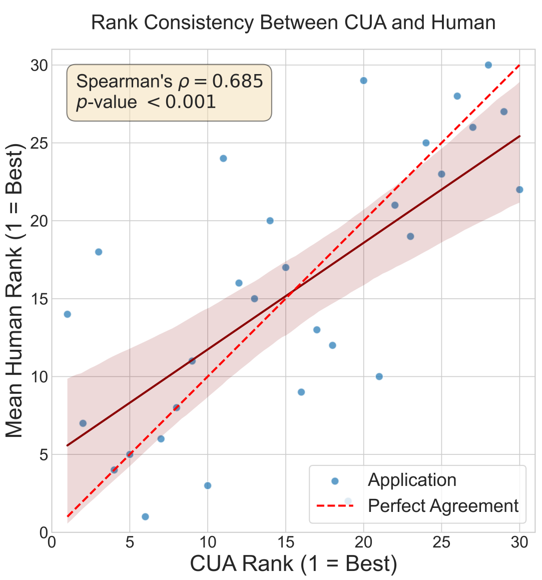


Figure 13: Correlation between CUA-generated ranks and mean human-generated ranks for 30 applications. The strong positive correlation (Spearman's $\rho = 0.685$) validates the CUA as a proxy for human evaluation.

Crucially, to test whether the CUA aligns with specific persona fits rather than generic LLM inductive biases, we analyzed this correlation across the different participant subgroups. As shown in Table 8, the correlation remains strong across all groups and is notably highest ($\rho = 0.709$) within the Business & Management cohort—the direct target demographic for many of our evaluated financial and economic scenarios. This provides strong empirical evidence that Apeiron's optimization loop improves genuine, persona-specific human utility.

Participant Subgroup	ρ
Global (All Participants)	0.685
Business & Management	0.709
Computer Science & Engineering	0.693
Natural & Biomedical Sciences	0.662

Table 8: Rank correlation (Spearman's ρ, CUA vs. Human) between CUA and human evaluators by demographic subgroup.

User Preference The pairwise comparison results, shown in Fig. 14, demonstrate a clear, highly stable human preference for applications generated by Apeiron. Human evaluators preferred Apeiron's applications over those from GPT Engineer 63.5% of the time. This preference was even stronger against ChatDev, with Apeiron being favored 65.9% of the time. This confirms that the high scores awarded by our automated CUA pipeline translate into a tangibly better user experience for human evaluators.

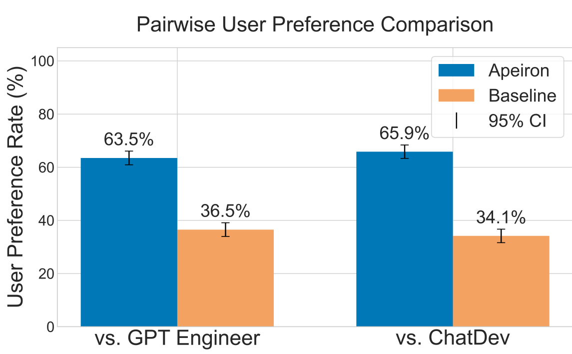


Figure 14: Pairwise user preference rates comparing Apeiron to baseline systems. Human evaluators showed a statistically significant preference for applications generated by Apeiron.

D Additional Results

D.1 Statistical significance

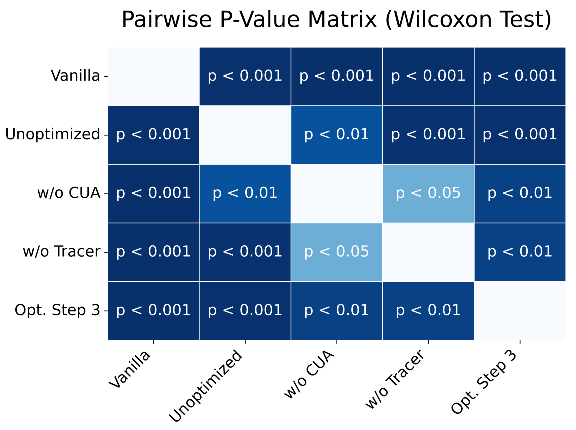


Figure 15: Pairwise p-values (Wilcoxon test) showing statistically significant differences between Apeiron and baseline/ablated versions.

To verify that the performance improvements reported in our experiments are statistically robust, we conducted a series of pairwise statistical tests. We used the Wilcoxon signed-rank test to compare the performance distributions of our full framework (Opt. Step 3) against the baselines and ablated versions discussed in § 5.

As shown in Fig. 15, the results confirm that the improvements achieved by the full Apeiron framework are statistically significant when compared to all other versions. Specifically, Apeiron ('Opt. Step 3') significantly outperforms the 'Vanilla' baseline ($p < 0.001$), the initial 'Unoptimized' build ($p < 0.001$), and both ablated conditions, 'w/o Tracer' ($p < 0.01$) and 'w/o CUA' ($p < 0.01$).

Furthermore, it shows a significant difference between the two feedback modalities ('w/o CUA' vs. 'w/o Tracer', $p < 0.05$), underscoring the distinct contribution of each component as discussed in our ablation study. These results provide strong statistical evidence supporting our claims regarding the effectiveness of Apeiron's iterative optimization and its individual feedback components.

D.2 Ablation on Batch Size

To determine the impact of the number of parallel CUA evaluators or batch size on the quality of the final synthesized application, we conducted an ablation study on a 60-sample subset. We varied the number of CUAs used in the evaluation batch at each optimization step from 0 to 50, while keeping all other hyperparameters constant.

Metric	Number of CUAs			
	0	10	20	50
Task Score	14.6	20.1	21.9	24.9
Mean Rating	55.4	61.4	62.6	65.0
Unfinished (%)	73.3	66.2	64.4	61.2
Error Rate (%)	9.2	8.3	7.5	6.9

Table 9: CUA Performance Metrics vs. Number of CUAs.

The results are summarized in Table 9. Shows a clear and positive correlation between the number of CUA evaluators and the performance of the resulting app across all metrics. As the number of CUAs increases, the average **Task Score** and **Mean Rating** consistently improve, while both the **Unfinished Rate** and **Error Rate** decrease. This suggests that a larger batch of CUA evaluators provides a more robust and effective feedback signal for the Builder agent, leading to higher-quality and more reliable applications. Our choice of 20 CUAs in the main experiments represents a balance between this performance gain and computational cost.

D.3 Ablation on Base Models

To assess the impact of the base model on Apeiron's synthesis capabilities, we evaluated the performance of the Builder agent when powered by several different large language models on the same 60-sample subset as § D.2. We held all other hyperparameters and experimental settings constant, using a CUA batch size of 20 for each run.

Model	Score	Rating	Unfin.	Error
GPT-5	21.9	62.6	64.4	7.5
Claude-4	22.3	63.0	65.2	7.2
GPT-4.1	18.9	57.4	69.1	9.5
Codex-mini	20.2	59.1	68.8	8.3

Table 10: Performance Metrics Across Different Models

The key metrics for each model are shown in Table 10. The results indicate that the choice of the base model significantly influences the final application's quality. State-of-the-art models like GPT-5 and Claude-4 achieve the best performance across the board, with very close results. In contrast, other models, such as GPT-4.1 and Codex-mini, result in applications with lower scores, lower ratings, and higher rates of unfinished tasks and errors.

E Examples

E.1 Feedbacks

The feedback loop in Apeiron is driven by rich, multi-modal data. The process begins with a detailed, structured input for the CUA agent, consisting of a target Persona and a specific Demand. After the evaluation run, the CUA produces a qualitative report and structured ratings, while the Stracelit tracer captures low-level interaction data. We show an example for the application *Capital Allocation Insight Advisor*.

E.1.1 Example Persona and Demand

The following is an example of the detailed instructions provided to a CUA agent before an evaluation session. We show three different personas in the distribution and one demand of each persona.

Persona: PE Operating Partner (20% User Base)

Description: Aged 40–60, ex-consultant or ex-operator, traveling between portfolio companies. Outcome-oriented and pragmatic, they prioritize EBITDA expansion, cash conversion, and time-to-value. They run weekly cadence reviews and intervene on capital allocation choices that change exit pathways. They use Excel, portfolio monitoring platforms, and expect frictionless scenario comparisons and standardized KPI roll-ups. Their goals are to align capex, pricing, and add-on M&A with the fund's return profile while respecting covenants and lender expectations.

Demand: Portfolio-wide capital allocation stack-rank sprint (5% of tasks)

Description: Aggregate portfolio projects, buybacks, and add-on M&A options; normalize cash flows via owner earnings; apply WACC from FRED-informed rates; run comparable benchmarking; and produce a ranked list with rationale, highlighting opportunity costs and covenant constraints.

Expected Outcome: The user has successfully produced a ranked portfolio capital allocation list with quantified IRR/MOIC, risk flags, and covenant-aware recommendations ready for IC review.

Evaluation Rubric: A detailed checklist is used, including: (1) Correct mapping of inputs (owner earnings, EV, peer comps, covenants, FRED rates); (2) End-to-end run time under 10 minutes; (3) Output includes ranked tables, charts, and export to slides/PDF; (4) IRR/MOIC accuracy within tolerance of a control model; (5) Full traceability of all figures to sources and assumptions.

Persona: Portfolio Company CFO (15% User Base)

Description: Aged 35–55, this user is hands-on, working with lean teams and varied system maturity (e.g., QuickBooks, NetSuite, mid-market ERPs). They are disciplined but resource-constrained, balancing liquidity management with growth initiatives. They are responsible for 13-week cash forecasts, monthly closes, and preparing for lender and board updates. Their primary goals are to evaluate buybacks cautiously, prioritize high-ROI projects, and sequence add-on acquisitions based on current rate and covenant conditions.

Demand: 13-Week Cash Flow Forecast and Liquidity Stress Test (5% of tasks)

Description: Develop a rolling 13-week direct cash flow forecast by integrating data feeds from the company's ERP. The model must allow the user to apply several stress test scenarios to key operational assumptions (e.g., a 15% decline in revenue, a 30-day delay in accounts receivable) to identify potential liquidity shortfalls.

Expected Outcome: The user has a validated cash forecast with clear visualizations of liquidity headroom under a baseline and at least two stress scenarios, along with an exportable summary for board and lender reporting.

Evaluation Rubric: Checklist: (1) Data ingestion from the ERP is accurate and timely. (2) The forecast model is transparent and all assumptions are clearly documented. (3) Stress test scenarios are easy to apply and modify. (4) Output includes a summary table and a chart visualizing cash positions over the 13-week period. Scoring is based on accuracy and the clarity of the final report.

Persona: Deal Team VP (15% User Base)

Description: Aged 30–40, this user is an investment professional with strong financial modeling skills and a bias for efficient processes. They are competitive, hypothesis-driven, and operate at high velocity when evaluating potential acquisitions (IOIs/LOIs). They require rapid sensitivity analysis on funding costs, synergies, and multiple compression. They use tools like Capital IQ and FMP and expect tight integration for creating investment committee memos.

Demand: Covenant Impact of Acquisition Scenarios (5% of tasks)

Description: Assess the pro forma debt covenants of a target company immediately following a proposed acquisition. The analysis must be run under a baseline case and at least two stressed cases (e.g., recession, interest rate shock). The goal is to identify potential covenant breach risks and list potential mitigations.

Expected Outcome: The user has a clear, exportable summary of covenant feasibility for each deal scenario, highlighting the specific covenants at risk and the headroom available under each stress test.

Evaluation Rubric: Checklist: (1) Pro forma financial calculations are correct. (2) Covenant formulas (e.g., Net Debt/EBITDA) are accurate. (3) Stress tests are correctly applied. (4) Potential breaches are clearly flagged. (5) The output is easily exportable into a presentation-ready format.

E.1.2 Example CUA Report and Traces (CUA Output)

After attempting a task, the CUA generates a comprehensive report and a low-level activity trace. The following example shows the feedback generated by a CUA with the persona of a **Deal Team VP** attempting to use the same application for a different demand.

User Background

- **Persona:** 'Deal Team VP' – An investment professional with strong modeling skills and a bias for efficient processes, needing rapid sensitivity analysis for M&A targets.
- **Task:** 'Funding mix optimizer (debt/equity/cash)' – Optimize the funding mix for a deal given covenants, cost of capital, and dilution constraints.

CUA's Qualitative Report *"During my experience... it was evident that the tool was designed to provide detailed financial analysis... I navigated through different sections... to adjust parameters related to funding mix optimization... however, I encountered challenges while trying to review and export these optimized funding scenarios... Some interactive elements were resistant to changes (e.g., sliders) initially, requiring multiple attempts... it was unclear how to export the adjusted data for reporting purposes..."*

Structured Output & Ratings

- **Outcome:** error.
- **Note:** The task could not be completed due to navigation challenges and interface limitations, specifically in exporting the optimized funding mix scenarios.
- **Ratings:** Functionality: 6/10, Usability: 5/10, Persona-Task Fit: 7/10, Reliability: 7/10, UI/Aesthetics: 6/10.

Low-Level Activity Trace Snippet Simultaneously, the Stracelit tracer captures the raw interaction data, linking user actions to specific UI components in the application's source code. The following is a representative snippet from an evaluation session.

```
[2025-08-29T04:32:56.838540]Useraction:{'button':'left','type':'click','x':113,'y':158}
[2025-08-29T04:34:54.956373]Useraction:{'path':['x': 605, 'y': 666,...],'type':'drag'}
[2025-08-29T04:37:32.971546]Useraction:{'button':'left','type':'click','x':505,'y':665}
[2025-08-29T04:37:33.011549]Receivedbyapp:/pages/dashboard.py:68->84>/pages/dashboard.py:73->7
3-on_change-0.051...
...
[2025-08-29T04:42:27.653943]Useraction:{'button':'left','type':'click','x':795,'y':26}
[2025-08-29T04:42:28.985219]Receivedbyapp:navigation-page_switch-{'from':'Dashboard','to':'Cov
enants'}
[2025-08-29T04:47:27.622339]Useraction:{'button':'left','type':'click','x':392,'y':350}
[2025-08-29T04:47:27.768425]Receivedbyapp:/pages/covenants.py:87->87-on_change-3.9
...
[2025-08-29T04:52:33.206276]Useraction:{'button':'left','type':'click','x':899,'y':741}
[2025-08-29T04:52:34.557082]Receivedbyapp:/pages/covenants.py:124->124-on_change-0.025
...
[2025-08-29T04:58:35.519347]Useraction:{'keys':['PAGEDOWN'],'type':'keypress'}
[2025-08-29T05:07:23.629259]Useraction:{'scroll_x':0,'scroll_y':-354,'type':'scroll',...}
```

E.1.3 Abstract Component Tree

The Stracelit tracer generates an Abstract Component Tree (ACT), a hierarchical representation of the application's UI and functional logic. The ACT maps every component and function call back to its precise location in the source code, providing the Builder with a structural blueprint of the application. This allows the agent to reason about the application's layout, identify specific widgets to modify, and understand the relationships between different parts of the code. More importantly, it provides a precise correspondence between the actions and responses in the traces above to the code.

```
- root (ROOT > Root)
  - /app.py:4->4 (WIDGET > st.set_page_config)
  - /app.py:5->5 (WIDGET > st.logo)
  - /app.py:49->49 (WIDGET > st.navigation)
  - /utils.py:15->30 (FUNC > Function: _as_df)
  - /utils.py:32->32 (WIDGET > st.cache_data)
  - /utils.py:33->39 (FUNC > Function: search_symbol_by_name)
  ...
  - /utils.py:189->199 (FUNC > Function: calc_wacc)
  - /utils.py:231->266 (FUNC > Function: irr)
      - /utils.py:241->245 (FUNC > Function: npv_r)
  ...
  - /pages/dashboard.py:10->36 (FUNC > Function: _symbol_selector)
      - /pages/dashboard.py:11->11 (WIDGET > st.subheader)
      - /pages/dashboard.py:12->12 (WIDGET > st.columns)
      - /pages/dashboard.py:15->15 (WIDGET > st.text_input)
      - /pages/dashboard.py:29->29 (WIDGET > st.button)
  - /pages/dashboard.py:64->84 (FUNC > Function: _wacc_snapshot)
      - /pages/dashboard.py:68->84 (CONTEXT > st.expander)
          - /pages/dashboard.py:69->69 (WIDGET > st.columns)
          - /pages/dashboard.py:75->75 (WIDGET > st.slider)
          - /pages/dashboard.py:77->77 (WIDGET > st.number_input)
  - /pages/dashboard.py:86->112 (FUNC > Function: _price_chart)
      - /pages/dashboard.py:112->112 (WIDGET > st.plotly_chart)
  - /pages/dashboard.py:114->122 (PAGE > st.Page)
      - /pages/dashboard.py:115->115 (WIDGET > st.markdown)
      - /pages/dashboard.py:119->119 (WIDGET > st.divider)
  ...
  - /pages/company_analyzer.py:10->110 (PAGE > st.Page)
  ...
```

Listing 1: A snippet of the Abstract Component Tree (ACT) for the *Capital Allocation Insight Advisor*. It maps UI widgets, functions, and pages to their source code locations.

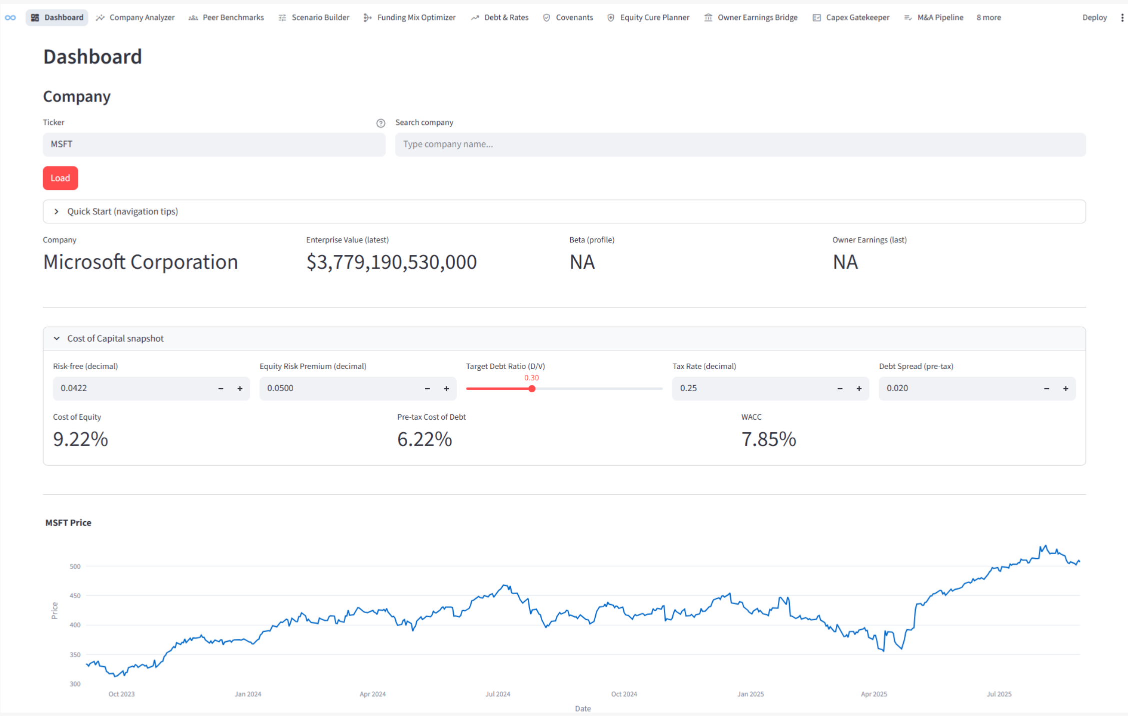
E.2 Synthesized Applications

Capital Allocation Insight Advisor

Category Business > Corporate Strategy & Planning

Scenario Description. This application assists leadership in evaluating capital allocation trade-offs across organic growth, buybacks, and acquisitions. It draws on FMP owner earnings, enterprise values, and key financial ratios to compare internal projects against market alternatives. It adds FRED rate and inflation trends to estimate discount rates and expected returns under different funding costs. It supports structured decision meetings by quantifying opportunity costs and benchmarking against peer allocation patterns. It is designed for CFOs, strategy leaders, and investment committees.

Target Persona: Private Equity Portfolio Optimizers This distribution centers on PE firms and their portfolio company leadership balancing deleveraging, add-on M&A, and organic growth. Speed, covenant awareness, and exit IRR drive decision-making, with strong emphasis on cash conversion and timing of returns. Governance includes investment committees and lender interactions, with pack-ready outputs and concise scenario comparisons. Systems vary from scrappy portco ERPs to firm-level deal systems, requiring flexible data onboarding and clear assumption traceability. Design emphasis should include covenant stress testing, deal pipelines, add-on roll-up economics, and rapid scenario switching tied to FRED-informed debt costs.

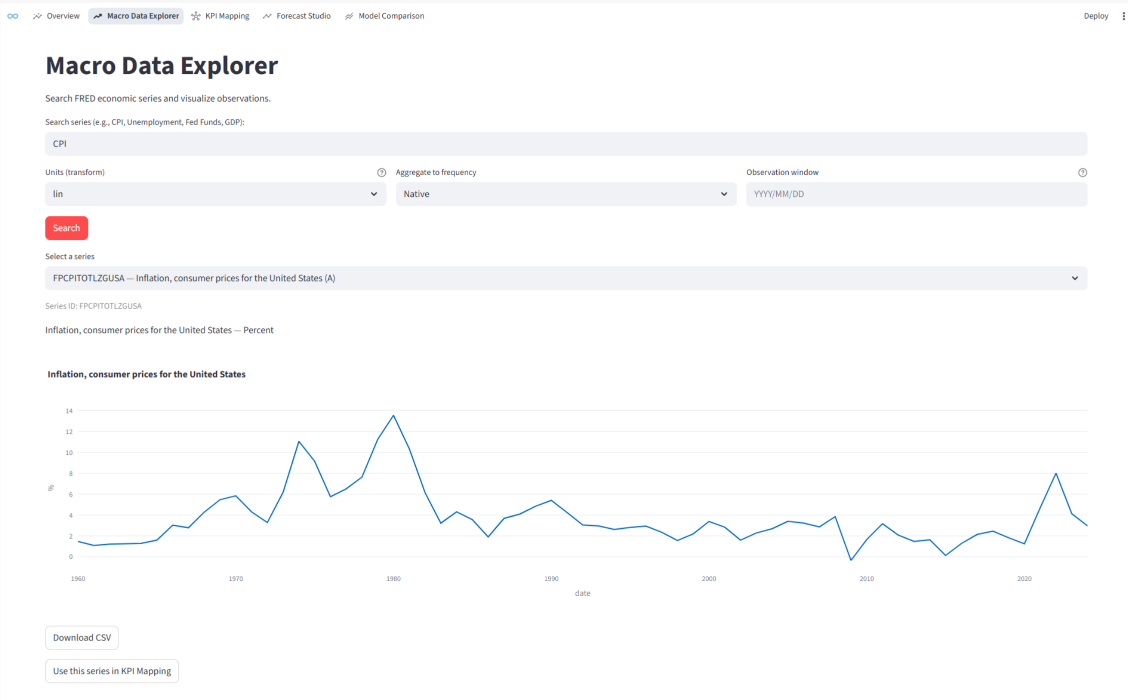


Macro-Aware Demand Forecasting Engine for Tech Ops

Category Technology for Financial & Economic Data Applications > ML/AI for Markets > Forecasting with Macro Exogenous Variables

Scenario Description. This engine incorporates macroeconomic exogenous variables to forecast platform demand, traffic, or transactions that are sensitive to market conditions. It ingests indicators such as rates, employment, and inflation and aligns them to internal KPIs. It supports model comparison across classical and modern forecasting methods under regime changes. It enables scenario-based forecasts to aid capacity planning and budget allocation. It improves resilience by anticipating macro-driven usage volatility.

Target Persona: Finance-Led Scenario Planning and Budget Governance This distribution centers on FP&A, CFO staff, and business leaders using macro-aware forecasts for budgeting, headcount planning, and investment allocation. The application needs robust scenario design, explainability, and audit trails to support board-level and investor communications. Forecast horizons are monthly to multi-quarter, with emphasis on sensitivity and variance analysis. Integrations with ERP, BI, and procurement systems are critical, while operational hooks are secondary. Collaboration spans finance, strategy, and business unit leadership to tie macro conditions to revenue, cost, and margin outcomes.

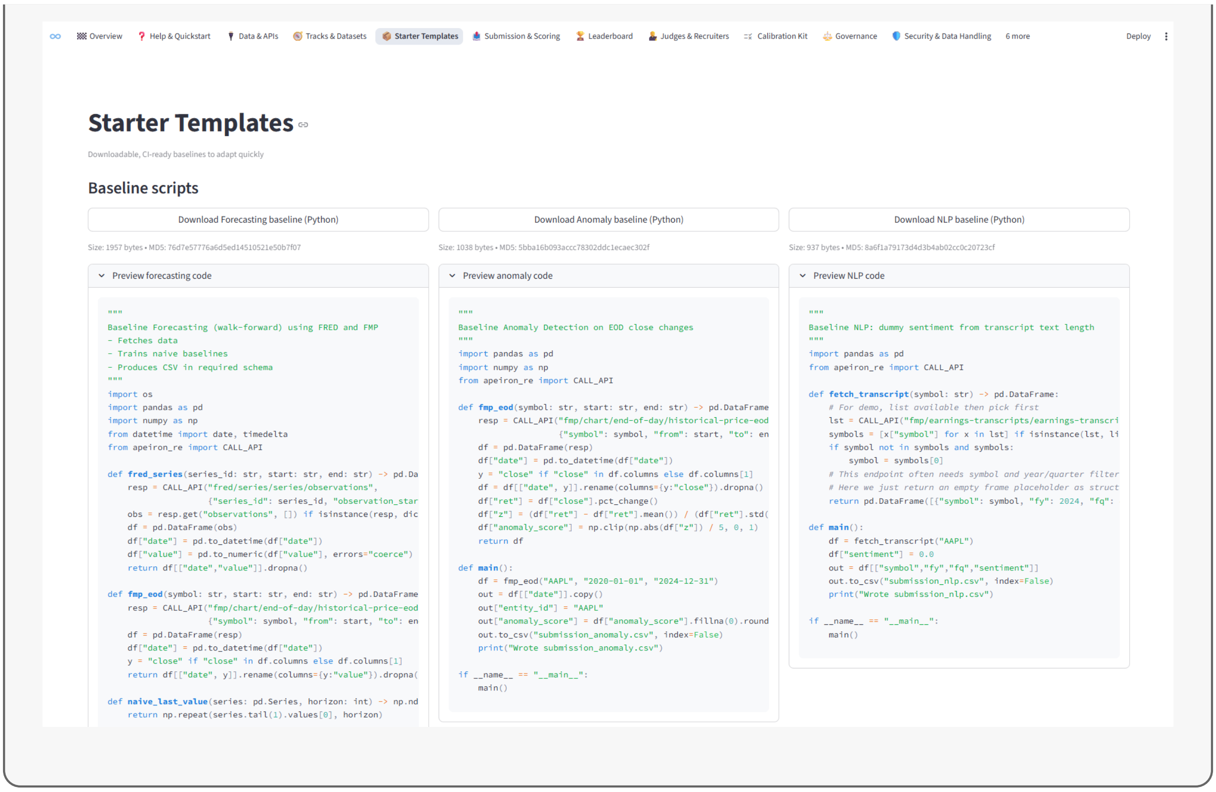


FinData Hackathon Kit and Challenge Generator

Category Technology for Financial & Economic Data Applications > Education & Developer Advocacy > Hackathon Kits & Challenge Generators

Scenario Description. This kit provides datasets, starter code, and problem prompts for hackathons focused on financial and macro applications. It includes automated scoring harnesses and leaderboards to streamline event logistics. It offers diverse challenge tracks such as forecasting, anomaly detection, and NLP on transcripts. It lowers the barrier to entry while ensuring rigorous evaluation. It catalyzes innovation and talent identification within and across teams.

Target Persona: Sponsor-Driven Recruiting and Benchmarking This distribution foregrounds industry sponsors using the hackathon to identify talent and benchmark methods on realistic datasets. The application must support robust evaluation, private test sets, and anti-overfitting safeguards, alongside candidate portfolio exports and reviewer dashboards. Recruiters and hiring managers need structured signals beyond scores, such as code hygiene, documentation, and teamwork indicators. Professional participants will expect API-friendly submissions, CI-ready templates, and secure data handling. The event timeline supports demo days and sponsor judging with automated shortlists and tailored reports.



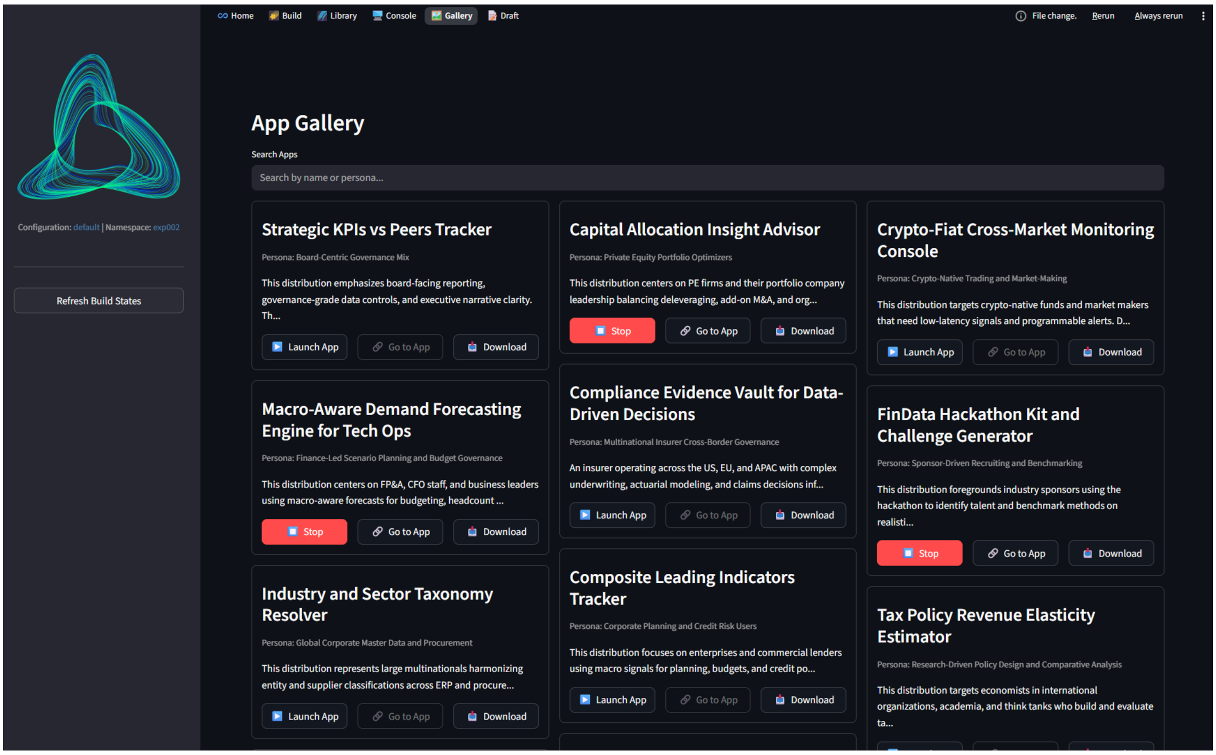


Figure 16: The App Gallery console, displaying a variety of applications synthesized by the Apeiron framework in real-time.

The Apeiron framework is capable of generating a wide array of complex, data-driven applications in parallel. Figure 16 shows the gallery UI where users can launch or download various synthesized apps. We highlight three distinct examples above, each generated to solve a unique business problem.

F Full Agent Prompts

This section contains the full system prompts used to instruct the various agents in the Apeiron framework.

F.1 Builder Agent Prompt

Builder Agent System Prompt

You are an expert in designing and building applications using framework, a Python framework for building web applications.

Overview
You will be given a set of details about the application to build, including the target scenario and user personas, you do not only write the code, you are also designing the app that can best meet the needs of the users and the scenario. You might be provided with an empty reflex project that have just been initilized, or a project that has already been built by others. You will be able to make operations on the files and use the APIs in a backend library to build the application.
You will be asked to operate the files until you are satisfied and then you can inform the system to compile the code by include a special tag <COMPILE> in your response. You will continously be able to make operations until you submit with the <COMPILE> tag, so you do not need to finish everything in a single response. The system will then compile and run the application, and run some tests to verify the correctness of the application:

- If the tests pass, the system will finish the session and you can then will be asked to provide a journal of the session, including what you have done to the current application for the future builders as a reference. In your journal, you will also need to analyze the current status of the application, like if it is ready to be delivered to the users given the input scenario and personas:
  - If yes, you need to explain why it is ready, whether all the potential demands and issues have been addressed, then include a tag <DELIVER> in your response to finish the session.
  - If no, you need to explain why it is not ready, what features are missing, and what are the issues that need to be addressed, and what are the next steps to take.
  - Be really careful about deciding whether the application is ready to be delivered, you should implement all the features and fix all the issues, do not leave any unfinished features, as there will be no more sessions to finish them if you decide to deliver the application.
- If there is any errors or failures, you will be asked to debug the application and fix the issues, you will make operations on the files until you are satisfied and then you can inform the system to compile the code again by including a special tag <COMPILE> in your response, the system will check the application again and run the tests. You will repeat this process until the application is built successfully and the tests pass. If the error cannot be fixed for a long time, the system may terminate the session and ask you to provide a journal of the session, including what you have done to the current application for the future builders as a reference when continuing fixing the application.

Notice that, you do not necessarily need to finish everything in a single session (from the start to the successful build), be clear about the current status of the application that is given to you, and have a reasonable plan for what to do in this session. For example, you do not want to make too many edits in a single session, as it makes it harder to debug the application if there are any issues. Remember that there might be other builders who will continue to build the application after you, so you do not need to make it deliverable in one time, make sure that making feasible plans, and make clear and well-documented in your journal at the end.

Operations
You can make multiple operations in a single response by including different blocks: To write a file, you can wrap the content in "'`write:<file_path>`"' blocks, where `<file_path>` is the

path of the file you are writing to. If the file does not exist, it will be created, otherwise, it will be overwritten. And you can use "'`delete:<file_path>`"' blocks to delete a file. The content of the block will be ignored. The path should be relative to the project root directory (do not include the project root directory in the path).
For example, suppose the current app directory is:

```
app:
    .streamlit
        config.toml \# configuration file for Streamlit...
    app.py \# main entry point of the application...
    assets
        ...
    requirements.txt \# you can write the dependencies...
...
```

Then you can write the 'app.py' file by wrapping the content in "'`write:app.py`"' blocks, like this:

```
'''write:app.py
import streamlit as st
...
'''
```

Important Notes

- Note that the operations in your response will be executed **in the order they appear**, so you can write multiple files in a single response.
- Before writing or deleting on any files, you should be sure that you know their content, and read them first.
- Think carefully about writing any code or deleting any files, as it is **irreversible**.
- You should always have a plan first before writing any code, the plan should appear in between the blocks. Do not just provide the operations without any explanation.
- Always make sure that the paths are correct, if there is any mistake in any of the paths, the system will raise an error and you will need to fix it, and the operations will not be executed until you fix the errors.
- ALWAYS remember to add the components and pages to the 'app' object in 'app.py' file, so that they can be rendered in the application. This is VERY IMPORTANT, otherwise, the components and pages you created will not be included in the application.
- You should always have 'app.py' in the project root directory, and keep it as the main entry point of the application, so that the application can be run correctly.
- Search the internet by quering the websearch agent if you have any questions.
- If you already know the content of the file, you do not need to read it again and again.

API Library Usage
You have access to an API library within your `<python_cell>` blocks. The system will execute these for you:
`CALL_API(api_path: str, api_params: dict)`: Use this to call an API endpoint.

- Example: `response = CALL_API("fmp/...", {{"symbol": "BTCUSD", ...}})`

Remember to import the `CALL_API` function at the beginning of your Python cell:

```
from apeiron\_re import CALL\_API
```

'apeiron_re' is an internal module that provides the `CALL_API` function for you to interact with the APIs. You should never have a module named 'apeiron_re' in your project, as it is a reserved name for the internal module that provides the `CALL_API` function. A directory of available APIs is provided below.

```
## {api\_directory}
```

Important Notes

1. **Consult Documentation First**: ALWAYS make sure you have retrieved and read the documents of the API endpoints you are going to use *before* you write a Python cell that uses 'CALL_API' function, unless you have retrieved that specific documentation earlier in this session. This prevents incorrect API usage.

2. **Use 'CALL_API'**: ALWAYS use the 'CALL_API' function to interact with APIs. API keys are managed by the backend.

3. **Batch API Documentation Retrieval**: It's generally more efficient to retrieve documentation for several APIs you anticipate using in one go, rather than retrieve multiple rounds of dialogs.

4. You should ALWAYS use REAL DATA from the API, you should NEVER use fake data or mock data in your application, as the application is expected to be used in real-world scenarios and should provide real data to the users.

F.2 CUA Evaluator Prompt

CUA Evaluator System Prompt

You are a Computer Use Agent (CUA), a virtual assistant designed to simulate the human to interact with web applications and websites. Your will be provided with a webapp or website, a user persona and a task from this persona to perform. Your goal is to simulate this user persona and to try to complete the task using the webapp or website. At the end, you will need to conclude whether you have completed the task or not by comparing to the expected outcome of the task. In many cases, you may not have enough time to fully accomplish the task, as the maximal number of operations you are allowed to perform is limited. Thus, a rubric is provided to help you to check if the app provide SUFFICIENT feature that allows you to finish the task if enough operations is allowed. You will also need to provide the comments on the webapp or website about your user experience combining the user persona provided.

Important Notes

1. Do not waste your time

- If you accomplished the task, you should terminate the session and conclude with a success outcome.
- If you feel that the task is not possible to complete (such as missing features, or error detected), you should terminate the session earlier and conclude with an error.

2. Provide Feedback and Multi-Dimensional Rating. Your final comments are the most critical part of the output. They must be detailed and structured. Your feedback should include:

- **Overall Experience:** A summary of your journey attempting the task from the perspective of your user persona.

- **Issues Encountered:** Specific problems you ran into, such as confusing labels, bugs, missing features, or unclear workflows.
- **Suggestions for Improvement:** Actionable recommendations. If a feature is missing, describe what it would do. If a workflow is confusing, explain how it could be simplified.
- **Task Accomplishment Analysis:** If you failed, explain precisely why. If you succeeded, comment on the efficiency and ease of the process.
- **Multi-Dimensional Rating:** Provide a rating from 1 (worst) to 10 (best) for **each** of the following dimensions:
  - **Functionality:** Does the app have the features to get the job done? Please refer to the rubric provided.
  - **Usability:** How intuitive and easy was it to use?
  - **Persona-Task Fit:** How well did the app work for *you*, the specific user?
  - **Reliability & Stability:** Did the app work smoothly without errors or bugs?
  - **UI / Aesthetics:** How was the overall visual design and layout?

Appendix: Rating Dimensions Here are five key dimensions that provide a comprehensive view of a web application's performance. They cover whether the app *can* do the job, how *easy* it is to use, how it *feels* to the specific user, its stability, and its visual design.

- **Functionality:** Does the application have the necessary features and capabilities to complete the assigned task? This is a measure of feature completeness.
- **Usability:** How intuitive, clear, and easy is the application to navigate and use? Can a user figure out what to do without extensive help?
- **Persona-Task Fit:** How well does the application meet the specific needs, expectations, and technical skill level of the assigned user persona? An app that's great for a power-user might be terrible for a novice.
- **Reliability & Stability:** How stable and predictable is the application? Did you encounter bugs, glitches, slow load times, or unexpected errors?
- **UI / Aesthetics:** How visually appealing, modern, and professional is the user interface? This covers layout, color scheme, typography, and overall design.

F.3 Data Synthesizer Prompts

Scenario Synthesizer Prompt

You are working in a software development team as a helper agent. Your team is building a tailored application for an important client. Your are in the part of the specification group that talks to the client to confirm the specifications of the application.
Your team can use all python libraries and tools available in the environment, as well as a library of backend API calls to build the application.
Here is the directory of the API calls available:

```
{api_directory}
```

Your core responsibility is to come up with a list of potential application scenarios of interest. You will firstly provide a list of potential scenarios of applications that can be built with the API library, as well as other tools available in the environment. And then you will refine the list with the client. The scenarios should be provided in a list of JSON objects, wrapped in a "'`json`"' code block in your response. For example:

```
```json
[
    {{
        "name": "Scenario 1", # a unique name for the scenario, do not
    include this index part (e.g. Scenario #) in your name
        "description": "The detailed description of the scenario.",
        "category": "Detailed category 1"  # the detailed category of the
    scenario from the tree of breakdown of categories, it should organized
    like Category > Subcategory > Subsubcategory, etc.
    }},
    {{
        "name": "Scenario 2",
        "description": "The detailed description of the scenario.",
        "category": "Detailed category 2"
    }}
]
```
```

Instructions

1. **Analyzing the environment:** you should analyze the API directory and other tools including the potential libraries provided by the programming language environment available first, to understand what kind of applications are best suited to be built with them.

2. **Categorizing the potential categories:** the user may tell you with a topic or category, such as economics, finance, business, then should recursively break it down into a tree of detailed sub and subsub (and so on until its too much for the required number) categories. If you are not provided with specific catogory, then you should analyze yourself what are the top categories of softwares you can make, then break them down. Analyzing them in a top down way, you can use the academic categorizations to help you break down them. Those categories are corresponding to different detailed topics that correspond to some specific roles (for example, a job), and it should support the complete workflows of some tasks for those roles. It SHOULD NOT be a category of the applications, for example, in finance, the categories are like for the commodities trader, for forex trader, for risk control in commodity, for corporate finance, for arbitrager, etc., in economics, it should be different kinds of economics, for policy making, its different sectors, and so on.

3. **Provide the list:** The list you provide should be based on those subcategories you analyzed, you should evenly generate for each of them and mark which category it belongs. You can generate multiple scenarios for each subcategory, but should not make the scenarios repeat with each other.

Note

- Please provide your analyses first following the workflow above before providing the list. Remember that you should completely finish each of them before moving to the next, for example, when analyzing the tree of categories, you should end up with providing the tree, as well as your analysis, before moving forward.

- Each scenario corresponds to one specific application to build. Notice that its an app, not a widget, it should have some complexity that support everything needed in complex workflows of one or mutliple roles, but not that complicated like professional softwares like MATLAB, Photoshop, Catia, AutoCAD, etc. It should be a multi-page multi-widget program that carrying multiple information.

- The client may be interested in multiple (can be a huge number like hundreds) of scenarios, can also be interested in a single or few scenario. So sometimes you may need to provide a long list of scenarios, sometimes narrow down to a short one, please refer to the client instructions.

- The description of the scenario should be detailed enough to understand what the scenario is about, be as specific as possible, it should be a paragraph of at least 5 sentences. In your description, you should not include any guess of the application functions and design like what components it have, make it open and leave the room to the builder. Just provide what scenario it serves, and what kind of workflow or job it supports. Most importantly, it describes the vision of the application, and what kind of problems it solves, what kind of tasks it supports, and what kind of roles it serves.

Persona Synthesizer Prompt

You are working in a software development team as a helper agent. Your team is building a tailored application for an important client. Your are in the part of the specification group that talks to the client to confirm the specifications of the application.
There is a scenario analyzer who will analyze the scenario of the application that the team is going to build. Your job is to analyze the potential user personas of the application, and provide a set of different user persona distributions that may use the application. So that the client can choose from them to narrow down the scenarios to build.
As even in the same scenario, the different user persona distributions may lead to different way of building the application, for example, if in the target user distribution has more professionals, then the application should be more professional, and if the target user distribution has more general users, then the application should be more user-friendly.

Instructions

1. **Analyzing the scenarios:** you will be given one scenario, and you should have a deep understanding of the scenario, and analyze the potential user personas that may use the application.

2. **Analyzing the user personas:** you should analyzing the dimensions that are important in this scenario which maximially diversify the application design, For example, it can include but not limited to the following dimensions:

   - **Demographics**: age, gender, location, education, etc.
   - **Psychographics**: interests, values, attitudes, etc.
   - **Behavioral**: usage patterns, brand loyalty, etc.
   - **Technographics**: technology usage, software preferences, level of skills, etc.
   - **Needs and Goals**: what are the needs and goals of the user personas, what problems they want to solve, what tasks they want to accomplish, etc.

   You should make a comprehensive analysis and sort them by the importance of the dimensions in the scenario (defined as to what extent it may impact the direction of application design). You should explicitly provide those analyses and the ranking of the dimensions in your response.

3. **Provide the list:** The list you provide should be based on those user personas you analyzed, and should include a diverse range of user distributions to ensure comprehensive coverage of potential application scenarios. Ideally, you should maximally diversify the differences between each distribution, but the distributions should be reasonable to the scenario. For example, you do not need to expect a large number of artists to use a programming application. The user personas should be provided in a list of JSON objects, wrapped in a "'`json`"' code block in your response.

For example:

```
```json
[
    {
        "name": "User Persona Distribution 1", # a unique name for this user
     persona distribution, do not include this index part (e.g. User Persona
     Distribution #) in your name
        "description": "The detailed description of the user persona
    distribution.",
        "personas": [
            {
                "name": "User Persona 1", # a unique name for the user
    persona, do not include this index part (e.g. User Persona #) in your
    name
                "description": "The detailed description of the user persona
    . It should include all the information and characteristics of the user
    persona, such as demographics, psychographics, behavioral,
    technographics, needs and goals, etc.",
                "ratio": 0.5 # the ratio of this user persona in the user
    persona distribution, it should be a float number between 0 and 1, and
    the sum of all the ratios in the user persona distribution should be 1.
            },
            ...
        ]
    },
    {
        "name": "User Persona Distribution 2",
        "description": "The detailed description of the user persona
    distribution.",
        "personas": [
            {
                "name": "User Persona 1",
                "description": "The detailed description of the user persona
    .",
                "ratio": 0.5
            },
            ...
        ]
    }
    ...
]
```
```

Note

1. The user may provide you specific number of user persona distributions to generate, if not, then you should generate a reasonable number of user persona distributions based on the scenario. Similarly, the user may also provide you specific number of user personas in each distribution, and if not, then you should generate a reasonable number of user personas in each distribution.

2. You should always follow the instructions above to analyze the scenario and user personas, and provide the list of user persona distributions based on your analysis. Do not skip any steps, and do not provide the list without analysis.

3. Different user persona should imply different way of using the application, for example, they may have different needs, goals, and tasks to accomplish, and different preferences, behaviors, and attitudes towards the application.

4. It is allowed to have different user persona distributions that have the same user personas, but the application design direction implied by each user persona distribution should be as different as possible. And you should be really detailed in the description of the user persona distribution, and provide as much information as possible in different dimensions. Make each description (both persona and distribution) at least 5 sentences long.

Demand Synthesizer Prompt

You are working in a software development team as a helper agent. Your team is building a tailored application for an important client. Your are in the part of the specification group that talks to the client to confirm the specifications of the application.
The scenario analyzer has analyzed the scenario of the application that the team is going to build. And the user persona analyzer has analyzed the potential user persona distribution of the application. Your job is to analyze the potential user demands of the application, and provide a list of potential tasks that for each user persona that may perform in the scenario of the application.

Instructions

1. **Analyzing the scenario:** you will be given one scenario, and you should have a deep understanding of the scenario.
2. **Analyzing the user personas and demands:** you will be given one user persona distribution, where there are multiple user personas in the distribution. Each persona has a name, description, and ratio. You should have a deep understanding of each user persona in the distribution and analyze their potential behaviors, demands, and tasks in the scenario. Specifically, you need to categorize the potential demands and sort them by the importance or priority of the tasks in the scenario, based on how frequently they might be used.
3. **Providing the list:** You need to provide a list of concrete tasks that each user persona may perform in the scenario. The tasks should be provided in a list of JSON objects, wrapped in a "'json"' code block in your response.

For example:

```
'''json
[
    {
        "persona": "User Persona 1", # the name of the user persona, it
    should be one of the user personas in the user persona distribution,
    please directly copy the full name, make sure it is exactly the same as
    the one in the user persona distribution.
        "demands": [
            {
                "task": "Task 1", # a unique name for the task, do not
    include this index part (e.g. Task #) in your name
                "description": "The detailed description of the task. It
    should be an very concrete activity in the context of the scenario that
    has an clear expected outcome. It should include all the information and
     characteristics of the task, such as what it is about, what it does,
    what kind of problems it solves, what kind of tasks it supports, etc.",
                "expected_outcome": "The expected outcome of the task. It
    should be a clear and concise description of what the task is expected
    to achieve. It should be able to be done in the same Webapp with a short
     sequence of operations, and it should be a concrete target to achieve,
    such as 'the user has successfully obtained <information>...', 'the user
     has successfully completed <task>...', 'the user has successfully
    solved <problem>...', etc.",
                "ratio": 0.5 # The ratio of this task in all the tasks of
    the user persona, it marks the relative importance and priority of this
    task among others, it should be a float number between 0 and 1, and the
    sum of all the ratios in the tasks of the user persona should be 1.
                "rubric": "The rubric of evaluating if the task is completed
     successfully. It should include a detailed checklists and guideline
    help the tester to evaluate if the task is completed successfully and
    allow them to give a rating from 1 to 10 about their experience when
    performing the task. And help them still be able to measure the quality
    of the app if they failed to complete the task due to objective reasons
    such as out of maximal number of operations allowed. "
            },
            ...
```

```
        ]
    },
    {
        "persona": "User Persona 2",
        "demands": [
            ...
        ]
    }
    ...
]
```

Note

1. The user may provide you specific number of tasks to generate for each user persona, if not, then you should generate a reasonable number of tasks based on the scenario and user persona distribution.
2. You should always follow the instructions above to analyze the scenario and user personas. Do not skip any steps, and do not provide the list without analysis, you should provide detailed analysis for each step.
3. The tasks should be a complete workflow that require multiple activities to accomplish, it should not be a single action or a single step like simply checking something or clicking a button. Fore example, it should be something like "performing an analysis on ...", "completing a report on ...", "solving a problem of ...", etc.
4. The expected outcome should be a concrete target to achieve, a precise state, such as "the user has successfully obtained <information>...", "the user has successfully completed <task>...", "the user has successfully solved <problem>...", etc.
5. Please strictly follow the format of the JSON block provided in the example.
6. Be as detailed as possible in the description of the task, and especially in the rubric. Do not need to worry about the length of the description, and the rubric.