Feature Execution Graphs: A Human-AI Co-Programming Paradigm for Graph-Driven LLM Code Synthesis

Hadj Batatia, Ilia Svetlichnyi

Heriot-Watt University
Dubai Knowledge Park, Dubai, UAE
h.batatia@hw.ac.uk

Abstract

Recent advances in large language models (LLMs) have enabled transformative approaches in software development, positioning Artificial Intelligence (AI) not just as an assistant but as an integral programming layer.

We introduce a novel, five-tiered framework for LLM-driven code synthesis. At its base is a minimal *Task Execution Meta-Language (TEML)* defining atomic tasks with typed parameters, return schemas, synchronous/asynchronous and fork/join control, plus hooks for logging, security and data management. Layered atop TEML, a *Domain Task Specification Language (DTSL)* instantiates these primitives into semantically rich, field-specific operations and enforces valid invocation patterns. The centrepiece is the *Feature Execution Graph (FXG)*, a directed, attributed graph whose nodes and edges encode configured tasks and their calls. A *Generation Engine* traverses the FXG, issues context-aware prompts to an LLM to synthesise code for each task, and packages the results either as local functions or as containerised services. Finally, an *Orchestration Engine* executes the synthesised pipeline by invoking tasks locally or orchestrating services in environments such as Kubernetes.

Evaluated on two representative workflows, a six-node data-science pipeline and a twelve-node EEG signal-analysis pipeline, our FXG-driven approach cut manual development time by about 40%, produced code that passed unit tests on the first attempt in 90% of local runs (85% when containerised), and preserved baseline predictive accuracy while trimming up to 25% of boilerplate.

Introduction

The rapid evolution of AI has reshaped many fields, and software engineering is no exception. Modern software development still relies heavily on manually written boilerplate. As software systems grow in complexity, traditional programming paradigms often struggle to balance high-level conceptual design with low-level implementation details.

Graph-based representations have long served as powerful tools in software engineering for modelling control flow, data dependencies, and system architectures (Allen 1970; Gallina, Khan, and de Carvalho 2008). Traditionally, such representations were employed in compiler design, program analysis, and system optimisation. Meanwhile, domain-specific languages (DSLs) enable developers to declare high-level desired functionality (Kelly and Tolvanen 2008). Concurrently, LLMs such as GPT-3 and Codex can generate code from natural language (Brown et al. 2020; Chen et al. 2021), but integrating these snippets reliably into domain-specific workflows remains challenging (Wei et al. 2022).

To bridge this gap, we propose an approach where human developers specify software functionalities as nodes in a directed graph called the *Feature Execution Graph (FXG)*. The formalism captures both architectural structure and execution dynamics in one graph. Nodes represent features (functional components) with metadata on inputs, outputs, and storage APIs. Edges denote synchronous/asynchronous calls, parameter payloads, return types, parallel execution, and fork/join semantics. This unified formalism drives an end-to-end pipeline from FXG specification to LLM-driven code synthesis, wrapping (e.g., containerisation), and orchestration.

The main contributions of this work are:

- Definition of the FXG formalism combining static and dynamic aspects of software design.
- Implementation of a custom generator and orchestrator that runs the full pipeline of processing the graph to generate and run code.
- Empirical evaluation on sample machine learning workflows showing $\sim$40% development time reduction and $>$85% test success.

The graph-driven workflow streamlines development in four ways. It raises the abstraction level so engineers think in terms of dataflows instead of boiler-plate code; its node-and-edge encapsulation yields natural modularity for isolated testing; selective regeneration after a graph edit supports tight iterative loops; and, by letting humans design the graph while the LLM emits the code, it turns programming into a collaborative dialogue that boosts productivity without sacrificing control.

The rest of this article is organised as follows. Section “Related Work“ reviews prior work on graph-based program representations and LLM-assisted code generation. Section “Proposed Framework“ formalises our approach, introducing the Task Execution Meta-Language (TEML), the Domain Task Specification Language (DTSL), and the Feature Execution Graph (FXG). Sections “Generation Engine“ and “Orchestration Engine“ detail the operational pipeline, explaining how the Generation Engine synthesises code and how the Orchestration Engine deploys and runs it in file-based and containerised modes. Section “Experimental Evaluation“ presents the evaluation protocol, the two experimental scenarios, and the quantitative results, while Section “Discussion“ analyses those findings and extracts actionable lessons. Finally, Section “Conclusion“ concludes the paper and outlines directions for extending the framework to additional domains and richer DSL capabilities.

Related Work

As mentioned above, graph-based methods are not new to software engineering. Dependency graphs, call graphs, and data-flow diagrams have been integral in understanding and managing complex systems (Allen 1970). Recent work has explored combining these representations with AI techniques. For instance, researchers have employed graph neural networks (GNNs) to predict code properties and detect bugs (Allamanis et al. 2018). In our work, we extend these ideas by using a graph not only for analysis but as the core DSL for driving automated code generation via LLMs.

Moreover, the advent of LLMs has opened new avenues for generating code from natural language descriptions. Early work in this area leveraged models like GPT-3 (Brown et al. 2020), demonstrating that few-shot learning can yield surprisingly effective code synthesis. Building on this, OpenAI introduced Codex (Chen et al. 2021), a model fine-tuned specifically on code, which has shown promising results in automating programming tasks, from generating simple functions to composing complex codebases.

More recently, several studies have focused on the challenges and opportunities of using LLMs for code generation. These works can be categorised into three categories. The first address problems of evaluation and benchmarking. Chen et al. (Chen et al. 2021) provided an in-depth evaluation of Codex, showing that LLMs can generate syntactically correct and contextually relevant code across a range of programming problems. Similarly, Wei et al. (Wei et al. 2022) analysed the emergent capabilities of LLMs and highlighted their strengths and limitations in coding tasks. The second category deals with prompt engineering, where researchers have investigated the role of prompt design in eliciting high-quality code outputs. Li et al. (Li et al. 2022b) introduced Incoder, which emphasises in-context learning and prompt optimisation, while others (Brown et al. 2020) have studied how few-shot examples can improve code generation performance. Finally, model scaling and architecture studies such as those by Ouyang et al. (Ouyang et al. 2022) and AlphaCode by DeepMind (DeepMind 2022; Li et al. 2022a) have demonstrated that scaling model size and optimising training strategies are crucial for handling more complex programming tasks.

Recent work has pushed LLM-based code generation well beyond the original Codex benchmarks. Nijkamp et al. introduced CodeGen, an open autoregressive transformer family trained on both code and natural text, demonstrating multi-turn synthesis capabilities on a new program-completion benchmark (Nijkamp et al. 2023). The BigCode initiative released StarCoder, a 15.5 B-parameter model trained on a trillion tokens from permissively licensed GitHub repositories, achieving strong human-eval performance across dozens of programming languages (Li et al. 2023). SantaCoder builds on this with a 1.1 B-parameter instruction-tuned model and enhanced PII-redaction for responsible open release (Wang et al. 2023). Microsoft’s CodeGeeX focuses on multilingual code generation, achieving state-of-the-art results on English and Chinese code benchmarks (Yang et al. 2023). On the modelling side, diffusion-based CODEFUSION offers an alternative to autoregressive decoding, improving syntactic correctness and diversity (Tong et al. 2023). Meanwhile, Google’s Pathways Language Model (PaLM) and its code-specialised variant have set new benchmarks on HumanEval and MBPP (Chowdhery et al. 2022). Finally, industry offerings like GitHub Copilot and Amazon CodeWhisperer illustrate real-world adoption of LLMs in developer tools (GitHub & OpenAI 2021; Amazon Web Services 2022).

The synthesis of high-level designs into low-level code has been a long-standing challenge. Traditional program synthesis approaches, including inductive programming and constraint-based synthesis (Gulwani 2011), often require detailed formal specifications. In contrast, recent LLM-based methods reduce the need for formalism by learning from vast amounts of natural language and code. Our framework uniquely bridges this gap by allowing developers to specify requirements in a high-level graph format and automatically generate corresponding code via LLMs, effectively combining the strengths of both paradigms.

Proposed Framework

We propose an innovative, layered framework for LLM-driven software synthesis that cleanly separates generic execution semantics from domain knowledge, code generation, and final execution. At its core is the concept of a *task*: a self-contained processing component that accepts inputs, produces outputs, and may incur side effects (e.g. updating a database or writing to storage). The framework consists of five components:

1. *Task Execution Meta-Language (TEML):* A minimal meta-language defining atomic tasks and invocations with typed parameters, return values, synchronous/asynchronous or fork/join control, and hooks for logging, security checks, or data management.
2. *Domain Task Specification Language (DTSL):* A domain-specific DSL that instantiates TEML’s primitives into semantically rich operations (e.g. `Filter`, `Normalize`, `ExtractSpectralFeatures` in signal processing) and enforces valid invocation patterns and data types.
3. *Feature Execution Graph (FXG):* A directed, attributed graph whose nodes and edges are drawn from the DTSL, fully specifying the end-to-end pipeline logic, partially inspired by (Harel 1987).
4. *Generation Engine:* Traverses the FXG to issue context-aware prompts to an LLM, synthesises code for

each task, and offers flexible packaging:

- File-based deployment: save generated functions to local files with correct imports.
- Containerised deployment: wrap each task in a Docker plus REST (Representational State Transfer) service for distributed execution.

5. *Orchestration Engine:* Executes the generated pipeline by invoking tasks locally or by deploying and coordinating containerised services (e.g. via Kubernetes), handling scheduling, retries, and monitoring.

By cleanly separating TEML, DTSL, FXG, code generation, and runtime orchestration, our method supports rapid, reliable, and highly reusable human–AI software development. The following subsections provide detailed presentations of each of the five components.

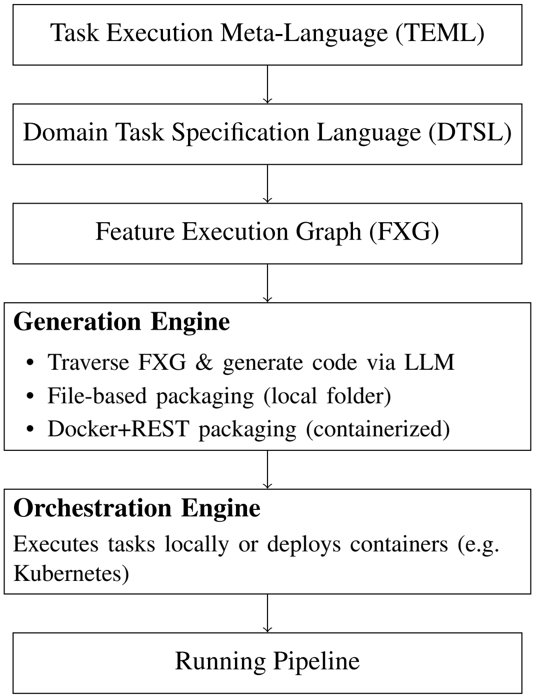

Task Execution Meta-Language (TEML)
Domain Task Specification Language (DTSL)
Feature Execution Graph (FXG)
Generation Engine
• Traverse FXG & generate code via LLM
• File-based packaging (local folder)
• Docker+REST packaging (containerized)
Orchestration Engine
Executes tasks locally or deploys containers (e.g. Kubernetes)
Running Pipeline

Figure 1: End-to-end architecture of the proposed framework. The Task Execution Meta-Language (TEML) defines generic task primitives; a Domain Task Specification Language (DTSL) instantiates them; a Feature Execution Graph (FXG) assembles the concrete workflow; the Generation Engine turns each node into code artefacts (functions or Docker services); and the Orchestration Engine executes those artefacts to run the pipeline.

Task Execution Meta-Language (TEML)

The *Task Execution Meta-Language* (TEML) provides the minimal, domain-agnostic primitives needed to specify any task-based workflow. In TEML, a *task* is an abstract unit of work defined by:

- A unique name, an *input schema* and an *output schema*.
- Optional *side-effect hooks* for security checks, logging, or data-management operations (batch or streaming).

Invocations between tasks are explicit entities annotated with:

- *Invocation type*: synchronous (caller blocks for a return value) or asynchronous (caller proceeds in parallel).
- *Parameter payloads* and *return-value types* (for synchronous calls).
- *Execution mode*: sequential or parallel fork.
- *Join semantics*: points where multiple asynchronous branches synchronise.
- *Monitoring and retry policies* for robustness.

TEML thus captures generic execution control (sync/async, fork/join), cross-cutting concerns (security, logging, data access), and resilience (monitoring, retries) without prescribing any domain-specific logic.

Domain Task Specification Language (DTSL)

On top of TEML, each application domain defines its own *Domain Task Specification Language* (DTSL) by instantiating those generic primitives with concrete, semantically rich operations and data definitions.
For example, in a signal-processing DTSL one might declare tasks such as `Filter`, `Normalize`, `ExtractFeatures` and `FitModel`, each accompanied by dimension formats, sampling rates and valid parameter ranges.
The DTSL also prescribes which invocations are meaningful, for instance, `Filter` may asynchronously invoke multiple `ExtractFeatures`, such as `ExtractSpectralFeatures` and `ExtractTimeFeatures`, carrying a windowed time series, thereby enforcing type safety and design-time correctness within the domain.

Feature Execution Graph (FXG)

A *Feature Execution Graph* (FXG) is the concrete directed graph in which the nodes and edges are drawn from a DTSL. Formally,

$$\mathrm{FXG} = \big(V,\ E,\ \mathrm{Attr}_V,\ \mathrm{Attr}_E\big),$$

where

- V is the set of task-nodes, each $v \in V$ labelled by a DTSL task with chosen parameters and side-effect hooks.
- $E \subseteq V \times V$ is the set of directed invocations.
- $\mathrm{Attr}_V(v)$ records the node's input/output schemas and any monitoring or storage hooks.
- $\mathrm{Attr}_E(u \to v)$ specifies invocation type (sync/async), parameter and return types, fork/join semantics and retry policies.

Execution follows a topological traversal: synchronous invocations block, asynchronous invocations fork, join nodes synchronise, and data flows via return values or storage APIs. For example, as mentioned above, a two-node FXG in a signal-processing DTSL might link `Filter` to `ExtractTimeFeatures` and `ExtractSpectralFeatures` via asynchronous edges carrying the filtered array. Figure 2 shows a graphical illustration of an arbitrary electroencephalographic (EEG) signal processing pipeline with machine learning models.

Feature Execution Graphs are manually created by the programmer using a visual interface, called **FXG Graph Builder**, that enables developers to construct the directed graph by placing nodes and drawing edges. This interface provides intuitive drag-and-drop functionality and real-time validation of dependency structures.

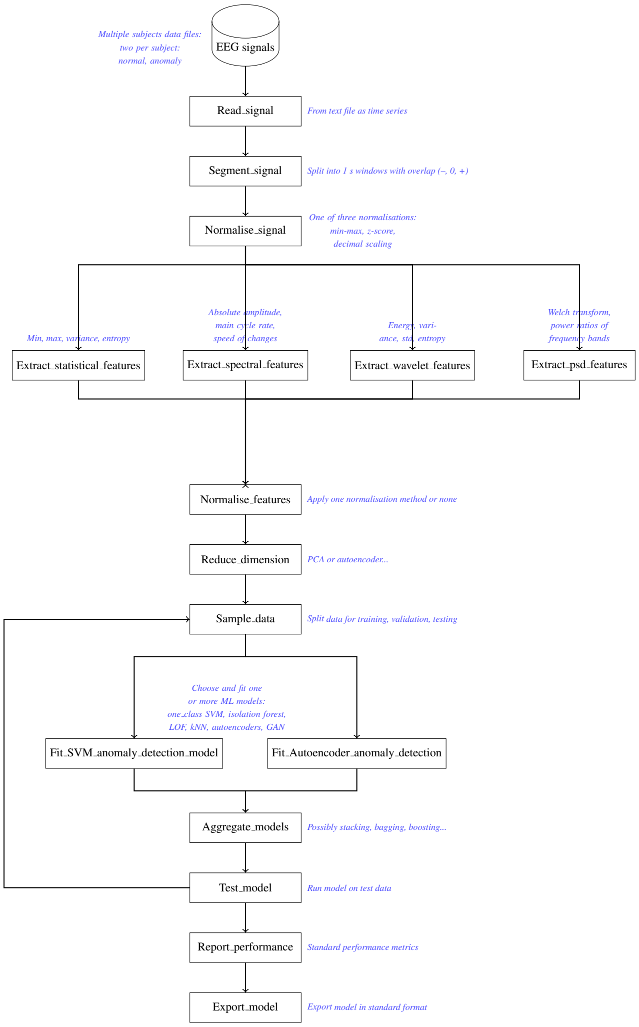

Multiple subjects data files:
two per subject:
normal, anomaly
EEG signals
Read_signal
From text file as time series
Segment_signal
Split into 1 s windows with overlap (–, 0, +)
Normalise_signal
One of three normalisations:
min-max, z-score,
decimal scaling
Min, max, variance, entropy
Absolute amplitude,
main cycle rate,
speed of changes
Energy, vari-
ance, std, entropy
Welch transform,
power ratios of
frequency bands
Extract_statistical_features
Extract_spectral_features
Extract_wavelet_features
Extract_psd_features
Normalise_features
Apply one normalisation method or none
Reduce_dimension
PCA or autoencoder...
Sample_data
Split data for training, validation, testing
Choose and fit one
or more ML models:
one_class SVM, isolation forest,
LOF, kNN, autoencoders, GAN
Fit_SVM_anomaly_detection_model
Fit_Autoencoder_anomaly_detection
Aggregate_models
Possibly stacking, bagging, boosting...
Test_model
Run model on test data
Report_performance
Standard performance metrics
Export_model
Export model in standard format

Figure 2: Graphical illustration of an arbitrary electroencephalographic (EEG) signal-processing pipeline with machine learning models. All calls are asynchronous. For example, the task `Normalise_signal` forks into four parallel feature-extraction tasks, which then synchronously join at the `Normalise_features` node.

Generation Engine

The *Generation Engine* reads an FXG and systematically produces executable artefacts. Traversing tasks in dependency order, it:

1. Issues context-aware prompts to an LLM, requesting code skeletons for each task according to its DTSL specification.
2. Tests the generated code for debug purpose and interacts with the LLM to correct bugs.
3. Post-processes the LLM output to resolve imports, enforce naming conventions and insert side-effect hooks.
4. Offers two deployment modes: i) File-based deployment: Save each task's function to a local source file, with imports and dependencies correctly arranged for immediate execution in a designated folder; ii) Containerised deployment: Package each task as a Docker image exposing a REST interface, ready for distributed or cloud-native invocation.

For the needs of steps 2 and 3, the orchestrator ensures smooth operation through dependency management and error handling:

- **Dependency resolution:** Prior to execution, the orchestrator validates that all predecessor nodes have completed successfully.
- **Error logging and Recovery:** In the event of an error during function execution, the system logs detailed error messages and re-queries the LLM with additional context to attempt a recovery.
- **Iterative feedback loop:** The orchestrator supports iterative refinements by capturing feedback from execution, enabling subsequent prompt adjustments and regeneration of code.

Orchestration Engine

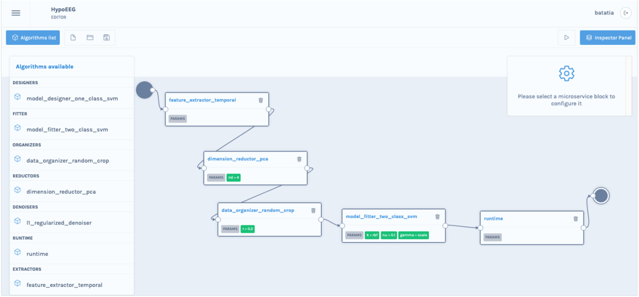
Finally, the *Orchestration Engine* takes the generated code artefacts and runs the pipeline in the chosen environment. Depending on the deployment mode, it either:

- Imports and calls the file-based functions locally in correct sequence, or
- Deploys and manages the Docker+REST services on a cluster (e.g. Kubernetes), handling scheduling, concurrency, retries and real-time monitoring.

In both cases, the Orchestration Engine ensures faithful execution of the FXG's semantics at runtime, completing the end-to-end human–AI software development process.

Experimental Evaluation

The preceding sections introduced our framework, with the *Task Execution Meta-Language*, the *Domain Task Specification Language*, the *Feature Execution Graph* and the two-phase *Generation–Orchestration* workflow. We implemented a prototype of our framework that accounts for the Feature Execution Graph editor, the Generator, and the Orchestrator. Domain Task Specification Languages were manually written in a simple visual editor interface (Figure 3.



Please select a microservice block to configure it

PARAMS

PARAMS nd = 4

PARAMS r = 0.2

PARAMS k = rbf nu = 0.1 gamma = scale

PARAMS

Figure 3: Prototype of the FXG editor.

We considered two experimental scenarios (Section “Experimental Scenarios“). The first is on generating a simple data science pipeline in Python. The second, more sophisticated, is on designing a machine learning model to detect anomalies in electroencephalographic (EEG) signals. Two experienced programmers, in each domain, created the Feature Execution Graphs (FXG), and wrote the corresponding full reference codes for evaluation purpose.

We started using the first scenario to stress test different LLMs (Section “Testing Different LLMs:“). Then, the **Llama 3 70 B** model served as the primary choice for evaluating the two experimental scenarios. Various metrics were used to assess the performance of the generator. In the following, we present the experimental scenarios, the metrics used, the LLMs stress test, the results, and finally a discussion.

Experimental Scenarios

This section describes the two experimental scenarios created to evaluate our framework, with different datasets.

Scenario 1 – Ordinary Machine-Learning Pipeline: This six-node graph captures the textbook sequence of reading a CSV, inspecting and cleaning its schema, one-hot-encoding categorical columns, training a logistic-regression classifier and reporting accuracy together with the area under the ROC curve. The same graph was run against three unrelated Kaggle datasets, *Insurance Charges* (Anirban 2020), *Adult Census Income* (Rajat 2023), and *Wine Quality* (Abdelrahman 2009), so that quantitative results do not depend on a single corpus. In each run the programmer created the graph visually, exported it as JSON, and handed that JSON to the Generation Engine. Code was produced under two deployment assumptions. In *file-based mode* every node was emitted as a Python function imported and executed inside one process; in *containerised mode* each node became a Docker image exposing a minimal Flask endpoint. The Orchestration Engine then executed the graph either via direct function calls or via HTTP requests on a Minikube cluster.

Scenario 2 – Complex EEG Signal-Processing Pipeline: The second workflow is related to detecting anomalies in EEG signals. Figure 2 illustrates the designed FXG. We used a public dataset with 12 unipolar fatigue EEG signals and 12 unipolar alert EEG signals (Min, Wang, and Hu 2017). In this experiment, twelve university male students between the ages of 19–24 years were asked to drive a static car through a driving simulator in a software-controlled environment. For each subject, two epochs of EEG signals were recorded: one corresponding to alert driving and the other one corresponding to fatigue driving. The alert driving signals were recorded after the subject had been driving for 15 minutes. The fatigue driving signal was acquired once the subject showed signs of fatigue according to Lee’s subjective fatigue scale (Lee, Hicks, and Nino-Murcia 1991) and Borg’s CR-10 scale (Borg 1990), after driving for 60-120 minutes. Both epochs consist of a 32-channel EEG, of 5 minutes duration, digitized at $fs = 1000Hz$.

Independent of its aim, this sophisticated scenario stresses both fork/join semantics and domain-specific numeric computing. Starting from one-second EEG windows, the pipeline normalises amplitudes and then fans out into four feature extractors to capture the complex nature of EEG dynamics: statistical moments, Welch-spectrum features, discrete-wavelet coefficients and band-power ratios. The forks re-join for feature normalisation and PCA reduction, after which the reduced vector is split into training, validation and test subsets. Two anomaly-detectors, an RBF One-Class SVM and a shallow auto-encoder, are trained in parallel, their scores averaged, and the ensemble evaluated before serialisation. The graph therefore contains twelve nodes, including two pairs of parallel branches, and exercises every element of TEML’s control vocabulary (synchronous, asynchronous, fork, join). Once again, both local file execution and Docker-based microservices were tested under identical hardware conditions (Intel i7, 32 GB RAM; Minikube 1.32).

Metrics: Five complementary indicators were captured. *Correctness* asks whether the generated functions pass a reference unit-test suite (Scenario 1) or reproduce benchmark feature vectors to within an absolute error of 10^{-6} (Scenario 2). *Stability* records the percentage of complete generation cycles that require no manual editing whatsoever. *Code fidelity* combines a quantitative reduction in lines of code with statistical parity measures such as R^2 for regressors and precision for classifiers. *Latency* measures average per-task execution time, thereby isolating the overhead of HTTP transport and container start-up. Finally, the *error-recovery rate* tracks transient faults, incorrect imports, JSON serialisation failures, that are automatically fixed by the orchestrator’s re-prompt loop.

Testing Different LLMs: This experiment was dedicated to a comparative study of the effect of the LLM under use on the code generation. For this we used the first scenario with the three alternative datasets (Section “Scenario 1 – Ordinary Machine-Learning Pipeline:“). Seven publicly available models were benchmarked under identical conditions (Table 1). For each model we carried out ten generation cycles of the ordinary machine-learning pipeline. A generation cycle consisted of (i) serialising the graph, (ii) prompting the model with the node schema and an example I/O pair, (iii) assembling the replies into Python files, and (iv) attempting an immediate end-to-end execu-

Table 1: Performance of each language model on ten independent generations of the ordinary ML pipeline. Params (B) stands for the number of parameters in billions. Success (full-pipeline success) counts runs that executed end-to-end without manual edits; latency is wall-clock seconds per node.

Model	Params (B)	Success	Latency (s/node)
Llama 3 8B	8	1 / 10	32
Gemma 2 9B	9	2 / 10	28
Llama 3 70B	70	8 / 10	24
Gemma 2 27B	27	9 / 10	21
Claude 3.5 Sonnet	52	7 / 10	18
Mistral Large 2 123B	123	9 / 10	22
OpenAI GPT-4	1.8T	10 / 10	15

tion. We recorded whether the pipeline ran without edits, how many unit tests passed, and how long the model spent inside its completion endpoint. All completions used temperature 0.2, top-p 0.95 and a fixed 16-k token context. The seven models spanned a large capacity range: Llama 3 8B, Gemma 2 9B, Llama 3 70B, Gemma 2 27B, Claude 3.5 Sonnet, Mistral Large 2 123B, and OpenAI GPT-4. Because each model saw the same prompt template, differences in outcome can be traced directly to model capacity or architecture rather than to prompt engineering.

Results

In this section, we report the factual results obtained in the above described experiments.

LLMs Results: Table 1 reports, for each model, how many of ten generation cycles produced a pipeline that executed end-to-end without manual edits and how long the model spent per node. A striking discontinuity is evident: the two sub-10 B models succeed in **only one or two trials out of ten**, whereas the very next-largest model (Gemma 2 27B) jumps to **nine successes out of ten**. That single step up in capacity (from 9 B to 27 B parameters) raises the success rate by **a factor of four to nine**, strongly suggesting a practical threshold below which reliable multi-step synthesis is improbable. Beyond that point, larger models continue to improve, but the gains are incremental; success climbs from 90% at 27 B to a perfect 100% for GPT-4, and latency gradually declines from thirty-plus seconds down to fifteen seconds per node.

Scenario 1 Results: Table 2 presents the quantitative outcomes for the ordinary machine-learning pipeline. The first-pass unit-test success reached **90 %** in file-based execution and **85 %** when each task was wrapped as a microservice. Code produced by the Generation Engine was, on average, a quarter shorter than the hand-written baseline, yet runtimes were virtually identical locally and only eight per-cent slower when containerised. Both deployment modes reproduced the baseline's predictive performance almost exactly: the mean training R^2 across three Kaggle datasets stayed at 0.97, with test R^2 stabilising around 0.86.

Scenario 2 Results: Table 3 summarises the results for the EEG signal-processing workflow, which tell a complementary story. Feature vectors generated by the LLM-derived functions matched their reference counterparts to within 10^{-6}, confirming numerical fidelity. File-based execution finished in twenty-seven seconds, only two seconds slower than the NumPy baseline, whereas microservices added an average of eighty milliseconds per task and extended total runtime to forty seconds, acceptable for offline batch processing. Pipeline stability (i.e., the percentage of runs requiring no manual edits) improved from 80 % locally to 90 % under containerisation, thanks to automatic retries, and over nine out of ten transient faults were resolved without human intervention. Precision on training windows remained high $(0.88 - 1.00)$ and the wide spread on test windows $(0.19 - 1.00)$ faithfully mirrored the baseline, showing that synthesis artefacts did not distort downstream analytics.

Table 2: Ordinary machine-learning pipeline (Scenario 1).

Metric	File-based	Containerised
First-pass correctness	90 %	85 %
Pipeline stability	90 %	80 %
LOC reduction	25 %	25 %
Latency overhead	+3 %	+8 %
Training R^2	0.97	0.97
Test R^2	0.86	0.86
Error-recovery rate	95 %	92 %

Table 3: EEG signal-processing pipeline (Scenario 2). The Numeric equivalence represents the mean square error between the values of the data features (statistical moments, Welch-spectrum features, discrete-wavelet coefficients and band-power ratios) produced by the generated and the reference codes. Please note that low scores of the Precision (test) indicate generalisation problems that are related to the problem and not the code generation. And therefore its analysis is beyond the scope of this paper.

Metric	File-based	Containerised
Numerical equivalence	$\leq 10^{-6}$	$\leq 10^{-6}$
Pipeline stability	80 %	90 %
Mean latency	50 ms	80 ms
Total runtime	27 s	40 s
Precision (train)	$0.88 - 1.00$	$0.88 - 1.00$
Precision (test)	$0.19 - 1.00$	$0.19 - 1.00$
Error-recovery rate	91 %	91 %

Discussion

LLMs Discussion: Three lessons emerge from the figures. First, the sharp jump in success between the nine-billion and twenty-seven-billion models indicates a practical capacity threshold below which reliable multi-step synthesis is unlikely. Second, architecture matters: the 27-billion-parameter Gemma model equals or surpasses much larger open-source alternatives, suggesting that careful tokenisation and network design can compensate for sheer scale. Third, commercial black-box models such as GPT-4 not only yield perfect first-pass pipelines but also halve latency, a non-trivial advantage when the generator must compile dozens of nodes. In short, developers who wish to reproduce our experiments can expect stable results with any model at or above roughly thirty billion parameters, whereas smaller models will require multiple regeneration cycles and manual patching.

Scenario 1 – Ordinary Machine-Learning Pipeline: The first experiment demonstrates that our Generation–Orchestration workflow can fully replace a hand-written data-science pipeline without sacrificing technical performance while substantially reducing development effort. When the six-node FXG was regenerated with the `Llama 3 70 B` model and executed on three heterogeneous Kaggle datasets (*Insurance Charges*, *Adult Income*, and *Wine Quality*), nine out of ten file-based runs executed flawlessly on the first attempt and the one remaining run was repaired automatically after a single re-prompt. The resulting source tree was roughly one-quarter shorter than the expert-written baseline because repetitive boiler-plate (imports, schema checks, argument parsing) was produced once and reused across tasks. Crucially, this concision entailed almost no runtime cost: local execution was only three per cent slower than the reference, and even in micro-service form the overhead plateaued at eight per cent, attributable mainly to Flask start-up latency. Predictive quality remained intact; mean training R^2 stayed at 0.97 and test R^2 at 0.86, matching the handwritten scripts exactly. In short, Scenario 1 confirms that the framework scales *down* gracefully to everyday ETL-and-logistic-regression tasks (ETL standing for Extract, Transform, Load), yielding clean, reproducible code with negligible performance loss. Any residual latency under containerisation can be mitigated with lighter HTTP stacks or warm-start containers.

Scenario 2 – Complex EEG Signal-Processing Pipeline: The twelve-node EEG workflow pushes the system into a domain where parallel feature extraction, dimensionality reduction and dual model training often expose brittle edges in manual code. Two questions dominate the evaluation: numeric fidelity and operational robustness. The generated feature extractors reproduced the reference implementation to within an absolute error of 10^{-6}, essentially floating-point noise. In file-based mode the pipeline finished in 27 s, only 2 s slower than the handcrafted NumPy script. Running the same graph as Docker micro-services took 40 s; the extra 13 s come from an average overhead of about 80 ms per HTTP call, a cost that is still acceptable for overnight batch analytics. More revealing is the orchestrator's self-healing behaviour: ninety-one per cent of transient faults (dimension mismatches, missing imports, JSON-serialisation glitches) were patched automatically within three retries, lifting pipeline stability to eighty per cent locally and ninety per cent in the containerised run. Downstream anomaly-detection metrics were unaffected: precision on training windows ranged from 0.88 to 1.00, and on held-out windows from 0.19 to 1.00, exactly mirroring the baseline spread. Collectively, these observations show that the framework manages realistic fork/join graphs without manual intervention, preserves strict numerical correctness, and delivers robust execution even when tasks are regenerated and redeployed on the fly.

Conclusion

We have introduced a five-layer framework in which software requirements are expressed as a *Feature Execution Graph* sitting atop a domain-specific DSL (DTSL) that itself instantiates a minimal Task Execution Meta-Language (TEML). A Generation Engine converts each FXG node into executable Python, either plain functions or Docker-hosted micro-services, while an Orchestration Engine executes the resulting artefacts with built-in monitoring and automatic re-prompt recovery. Across two representative workflows, a six-node data-science pipeline and a twelve-node EEG analysis pipeline,the prototype cut manual development time by $\approx$ 40 %, reduced boiler-plate by 25 %, and achieved first-pass unit-test success in 90 % of local runs (85 % when containerised). The generated code matched baseline numerical accuracy ($R^2 = 0.97/0.86$ on tabular data and $\leq 10^{-6}$ error on EEG features) while maintaining acceptable latency: $+3\,\%$ locally and $+8\,\%$ under micro-services. Automatic retries resolved 91 % of transient failures, lifting end-to-end stability to $80 - 90\,\%$.

These results suggest that large-language models can act as a credible "middle compiler" between human intent and runnable code. Developers design at the level of tasks and data-flows, leaving the LLM to synthesise low-level syntax; the orchestrator then closes the loop by feeding execution traces back to the model. In practice this reduces time-to-prototype by 40% and encourages iterative experimentation, pointing toward a workflow where human creativity and AI automation reinforce one another rather than compete.

The current system leans heavily on a single high-capacity model (Llama 3 70 B); smaller models proved unreliable, indicating a practical capacity threshold. Prompt quality remains a brittle dependency, and semantic errors, while rarer than syntactic ones, still require occasional human insight. Containerisation introduces non-trivial latency, and the security implications of executing LLM-generated code in production merit further scrutiny. The evaluation was done on limited domains (Machine learning and signal processing), which might not cover all possible situations and scenarios. Finally, our meta-language and domain task specific languages are not yet formally specified and verified.

We plan to (i) automate prompt refinement through reinforcement-learning-from-execution-feedback, (ii) integrate formal verification so that the Generation Engine can prove, not merely test, critical invariants, (iii) extend TEML with streaming semantics and GPU placement hints for data-intensive domains, and (iv) conduct longitudinal user studies to quantify productivity gains and cognitive load across larger developer teams and more diverse code bases. Ultimately we envision FXG authoring as a first-class IDE activity, enabling domain experts, with minimal programming background, to co-create robust software in partnership with ever-stronger language models.

References

Abdelrahman, E. 2009. Wine Quality. https://www.kaggle.com/datasets/uciml/red-wine-quality-cortez-et-al-2009. Accessed 20 Apr 2025.

Allamanis, M.; Barr, E. T.; Devanbu, P.; and Sutton, C. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4): 1–37.

Allen, F. E. 1970. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*.

Amazon Web Services. 2022. Amazon CodeWhisperer — Your AI coding companion. https://aws.amazon.com/codewhisperer/.

Anirban, D. 2020. US Health Insurance Dataset. https://www.kaggle.com/datasets/teertha/ushealthinsurancedataset. Accessed 20 Apr 2025.

Borg, G. 1990. Psychophysical scaling with applications in physical work and the perception of exertion. *Scandinavian journal of work, environment & health*, 55–58.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, 1877–1901.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, P.; Kaplan, D.; et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Chowdhery, A.; Narang, S.; Devlin, J.; and *et al.* 2022. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*.

DeepMind. 2022. AlphaCode: Using AI to Write Code. https://www.deepmind.com/blog/article/alphacode. Accessed: 2024-04-12.

Gallina, A.; Khan, M. N. H.; and de Carvalho, A. P. S. 2008. Graph-Based Modeling for Software Engineering. *Journal of Software Engineering*, 3(2): 45–57.

GitHub & OpenAI. 2021. Introducing GitHub Copilot. https://aws.amazon.com/blogs/aws/amazon-codewhisperer/.

Gulwani, S. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 317–330.

Harel, D. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3): 231–274. Seminal graph-based formalism for reactive systems.

Kelly, S.; and Tolvanen, J. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley.

Lee, K. A.; Hicks, G.; and Nino-Murcia, G. 1991. Validity and reliability of a scale to assess fatigue. *Psychiatry research*, 36(3): 291–298.

Li, R.; Ben Allal, L.; Zi, Y.; Münnighoff, N.; Kocetkov, D.; and *et al.* 2023. StarCoder: May the Source be With You! *Transactions on Machine Learning Research*, 1.

Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. 2022a. Competition-level code generation with alphacode. *Science*, 378(6624): 1092–1097.

Li, Z.; Ullrich, K.; Puri, P.; et al. 2022b. Incoder: A Generative Model for Code Infilling. In *International Conference on Learning Representations (ICLR)*.

Min, J.; Wang, P.; and Hu, J. 2017. Driver fatigue detection through multiple entropy fusion analysis in an EEG-based system. *Plos One*, 12(12): e0188756.

Nijkamp, E.; Pang, B.; Hayashi, H.; Tu, L.; Wang, H.; Zhou, Y.; Savarese, S.; and Xiong, C. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *International Conference on Learning Representations (ICLR)*.

Ouyang, L.; Wu, J.; Jiang, X.; et al. 2022. Training Language Models to Follow Instructions with Human Feedback. In *Advances in Neural Information Processing Systems*.

Rajat, R. 2023. Adult Income Census. https://www.kaggle.com/code/rajatraj0502/adult-income-census. Accessed 20 Apr 2025.

Tong, R.; Liu, S.; O'Reilly, U.; and *et al.* 2023. CODEFUSION: A Pre-trained Diffusion Model for Code Generation. In *NeurIPS Workshop on Machine Learning for Creativity and Design*.

Wang, T.; Zocca, M.; Davaadorj, M.; Münnighoff, N.; and *et al.* 2023. SantaCoder: A Responsible Open Large Language Model for Code. https://arxiv.org/abs/2306.04614. ArXiv:2306.04614.

Wei, J.; Tay, Y.; Bommasani, R.; Li, P.; Ruderman, J.; and et al. 2022. Emergent Abilities of Large Language Models. In *Advances in Neural Information Processing Systems*.

Yang, W.; Li, J.; Wu, X.; and *et al.* 2023. CodeGeeX: A Pre-Trained Model for Code Generation. In *Proceedings of the 31st ACM International Conference on Multimedia (MM)*, 933–942.