PULL DOWN COMPLEXITY WITH KUBRICK

PROGRAMMING LANGUAGE AND SYSTEM TO FACILITATE HUMAN DEVELOPERS AND AUTOMATONS

Giancarlo Frison - gfrison.com

ABSTRACT

AI tools are now the horsepower of computer programming. They are generally great for writing glue-code and integration tasks, probably less than ideal for complex problems on complicated programming settings. What could be the reasons that prevent generators on full-scale adoption?

I am implementing a declarative programming language that facilitates the synergy between automatons and humans on software development by forcing AI tools to generate intuitive code and to allow human operators to understand what is in there. It is an attempt to lower the barriers by simplifying the programming experience.

For giving you some reasoning, I will touch various aspects that includes cognitive aspects of problem-solving applied to programming and the role of AI tools such as LRMs on software development. I will provide arguments for stating that the accidental complexity of the programming system may worsening the performance of AI generators as well as it affects human developers.

Started as a *data programming* language for pairing database queries with programming controls, Kubrick may evolve in an advance integrated programming environment with the footprint of a *Jupiter notebook* and the immediacy of a *spreadsheet*. It lets agents to focus on what they want to achieve making easy things easily without expressivity losses.

To mitigate the lack of solution productivity in generators - the way that an optimizer generates deriving combinations of solutions from a set of given axioms - I extended the language for helping agents to cover *combinatorial problems*. Think of answer set programming (ASP) but with an eye on integration and usability.

This is an ongoing project that was the topic of my Master's dissertation *"Programming Language and System for Enhancing AI-Assisted Software Development"* I defended last December 2025, and it summarized several insights I gathered during my experience in the field. I just recently started to open source it and I will gradually share it on GitHub.

Giancarlo Frison

COGNITIVE ASPECTS OF PROGRAMMING

AI tools are everywhere. In every domain it is possible to find features that can benefit of AI capabilities. If we exclude certain high-risk sectors - nuclear plants, aviation control - AI is already assisting human labour, and the pace of adoption will certainly accelerate.

Language models were conceived for NLP tasks and trained with all sort of available text, so it is quite natural to think about their application in programming code generation. After all, programming is text-based for being written and read like any other text. Is it then reasonable that LRMs exhibits appreciable programming skills as they do on generating phrases?

With AI Tools it is not so bizarre to ship live entire applications in hours rather than weeks, but their effectiveness may fall short on accomplishing what prompters want them to do. Complaints regarding GenAI performances are summarized in:

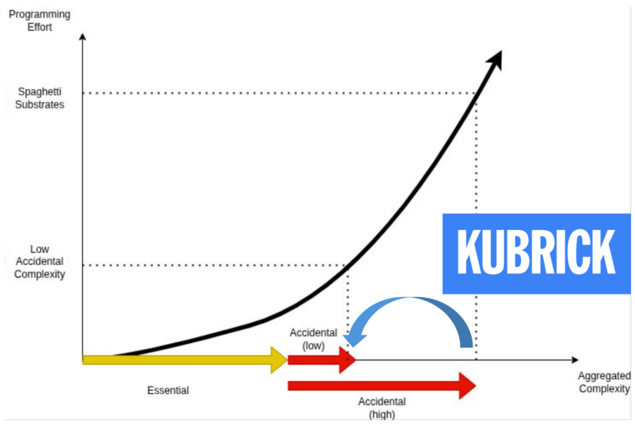
- Problems with multi-step reasoning.
- Struggles with mutable states and side effects.
- Shallow code understanding.
- Fail to meet requirements.
- Generate complicated code.
- Despite same prompt, they generate different code.
- Despite different prompt, they generate the same code.

Building a program essentially means to face it from two distinct sides: the **problem domain** and the **solution domain**. Understanding the problem to solve is the most important task, and if it is more or less difficult to grasp, it is related to its *essential complexity*. On the other hand, the complexity of the solution domain is referred to as *accidental complexity*, and it is introduced by the ecosystem necessary to implement the solution.

> *"If I had an hour to solve a problem, I'd spend 55 minutes thinking about the problem and 5 minutes thinking about the solution"* - A. Einstein

While increasing the complexity of a system absorbs more of the engineer's working memory, it also increases the perplexity an LRMs on solving the same task. Perplexity measures how unexpected a token is to the LRMs, and that means that the accidental complexity negatively impacts the generation of code, with more chances of introducing bugs not only by developers but also by automatons. Humans and the artificial agents show a significant positive correlation when faced with similar alienating settings.

The programming system consists of several *substrates* that include libraries, tools, and external systems, and when it increases the programming effort increases exponentially. While the essential complexity isn't negotiable, the accidental one must be kept at the lower level possible.



Many difficulties automatons and developers shows on generating code might be due to a common root: the accidental complexity carried by the programming system. The intuition behind this project is to address this issue from its foundations.

HOW TO REDUCE COMPLEXITY

Do you want to reduce accidental complexity and favor agents on programming? Be inspired by what teachers do for preventing plagiarism through AI, and apply the opposite strategies for making programming easier:

UNIFORM EXPERIENCE

From the point of programming experience, what is daunting for agents - in terms of increasing perplexity, and ultimately on increasing bugs - is the heterogeneity of the ecosystem. Different paradigms for configuration, data access, service orchestration, remote procedure calls, testing and deployment. Each of those aspects requires specific skills and knowledge that increases accidental complexity. A proper programming environment should nullify those frictions by providing a consistent an uniform way to interact with it.

OPEN AUTHORSHIP

Software development often involves a hard separation between programmers and users. I believe that accidental complexity prevents final consumers to be empowered to change the software applications. Are there already some examples that allow that?

I think everybody has at least once used a spreadsheet application. One thing that's very clear to users is that the spreadsheet does not have really a separate environment for programming and for use. A spreadsheet can be modified at any time by modifying the data or the formulas it contains.

Notebooks like Jupyter or Google Colab allow users to naturally split problems into smaller pieces, solve them individually with an immediate feedback. Those approaches lower the accidental complexity and Kubrick aims to combine both.

SELF-SUSTAINABILITY

> Self-sustainability refers to the extent to which a system's behavior can be changed without having to step outside to a lower implementation level - *Jakubovic 23*

The most predictor of a low-code platform's success is the ability to change the system's behaviour from the deepest layers. This would be achieved by introducing *macros*, programming fragments intended to generate programs inside the programming system itself. Why not being inspired by traditional languages like *Lisp* for that? It has stood the test of time with great honor, thanks also to its homoiconic nature that enables to write macros easily.

LOGIC + FUNCTIONAL

If we consider AI as an assistant, the programmer double-checks that what has been generated satisfy explicit and implicit requirements. Software code is more read than written and the immense capacity of the generators to flush out large quantity of code can easily saturate human scrutiny, urging the necessary of a clear, concise and easy language for encoding programs. We need to let agents to express *what* they want to achieve more than *how* to achieve it and remove the need of boilerplate code. Immutability, control over side-effects, unification, pattern-matching can definitely help on dragging down complexity.

RELATION ALGEBRA

I think one of the main complexity drivers is the impedance mismatch between query and programming languages. Those idiosyncratisms are usually mediated by ORMs frameworks but their slippery slope can trigger more problems than they solve. Relation algebra is the basic foundation of database theory and when combined with programming constrols, it provides a smooth experience for data-programming.

COMBINATORICS

When programmers encounter new code, they *actively simulate* the program's behaviour in their head and create mental models of its structure and logic. This is why programming is more than a logically demanding task with significant implications on how people interact with code. This is confirmed by the use debugging tools, syntax highlighting, code formatting, visualization applets; are all there for supporting the brain's reliance on logical simulations.

The attitude on playing code behaviours confirms that intelligence can't be diverted from the ability to search through a potential infinite combination of concepts and rules. This is in summary the idea of productivity which refers to the compositional generation of optimal propositions from a valid set of grounded statements.

Do automatons excel on that? The building blocks of LRMs lack of intrinsic search, though efforts have been applied on forcing recursive reiterations on their conclusions for minimizing hallucinations. How would be possible to inject intentional search where it is lacking? This is why combinatorial programming can boost applications' intelligence by commoditizing optimized solution search.

MAKE EASY THINGS EASILY

…AND COMPLEX THINGS DOABLE.

Not a single artefacts comes out of the vacuum and Kubrick is not an exception. I've been inspired by a multitude of established ideas on re-arranging them and creating new ones. From **Prolog** I took the variables' unification, the automatic pattern matching on method's activation and the success/failure assertions for validating method invocations, typical of logic programming. From **Julia** I've borrowed the fundamental recursive data structures for combining choices, named tuples and sequences. From **ASP** I've taken the *stable model* semantics for reasoning on alternative solutions and *choices*, the alternative values.

A COMBINATORIAL USE CASE

Central to la language is the recursive data type that include sequences, named tuples and choices:

```
grocery orange type→fruit price→1.2,
  rating→5 origin→italy;spain
```

Choices are alternative values (or expressions) to say that the fruit can either be 1st, 2nd, bio or local quality. Named tuples are key-value pairs and sequences are ordered collections, and any element can be a nested data structure.

Let's assume we need to create purchase lists with only one item per type and grouped by origin:

```
cart Item ⊣1^Origin,1#Type|
  grocery Item type→Type origin→Origin
```

You need to filter only purchase below 30€:

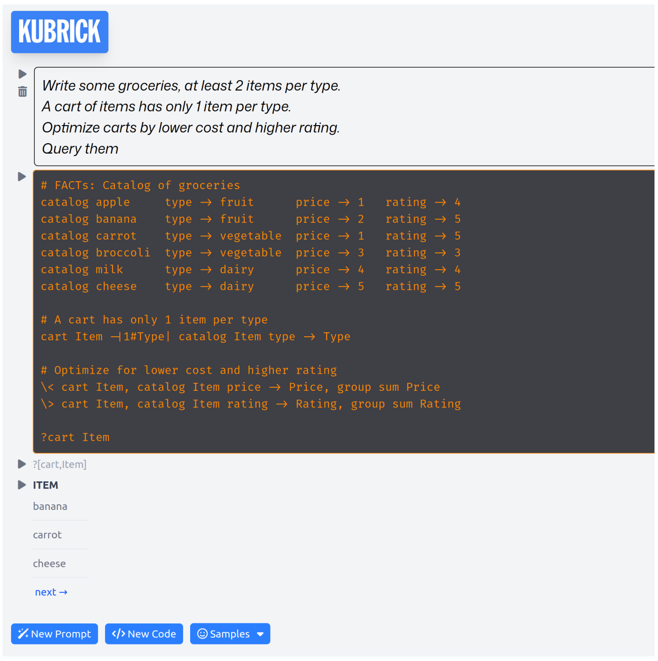
```
\ cart Item, grocery Item price→Price,
  Tot→(group sum Price),
  < Tot 30
```

and get the best combination of items that *maximizes* product's ratings and *minimizes* the total cost. View it as a sort of multi-objective Pareto optimization:

```
\> cart Item, groupcery Item rating→Rating,
   group sum Rating
\< cart Item, grocery Item price→Price,
   group sum Price
```

USER EXPERIENCE

The prototype's interface resemble a notebook environment with cells for writing code, for AI prompting and for visualizing results. While the program generation could be delegated to external LRMs, the program execution is performed locally in the browser. This approach combines the divide/conquer of notebooks, the immediacy of spreadsheet applications and the assisted program generation.



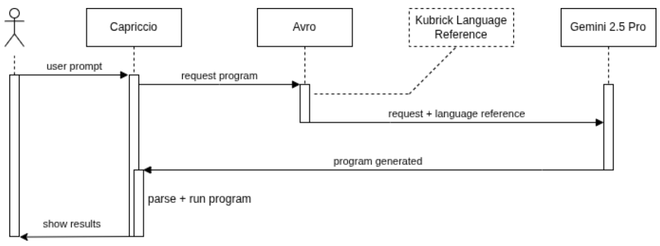
Write some groceries, at least 2 items per type.
A cart of items has only 1 item per type.
Optimize carts by lower cost and higher rating.
Query them

```
# FACTs: Catalog of groceries
catalog apple     type → fruit      price → 1   rating → 4
catalog banana    type → fruit      price → 2   rating → 5
catalog carrot    type → vegetable  price → 1   rating → 5
catalog broccoli  type → vegetable  price → 3   rating → 3
catalog milk      type → dairy      price → 4   rating → 4
catalog cheese    type → dairy      price → 5   rating → 5

# A cart has only 1 item per type
cart Item ⊣1#Type| catalog Item type → Type

# Optimize for lower cost and higher rating
\< cart Item, catalog Item price → Price, group sum Price
\> cart Item, catalog Item rating → Rating, group sum Rating

?cart Item
```

```
?[cart,Item]
ITEM
banana
carrot
cheese
```

PROCESS FLOW

When the user submit the prompt, the web application (Capriccio) delegate a specialized module (Avro) that augments the user prompt with Kubrick language reference documentation and forward the enriched request to the AI service. The generated code is then executed by the web application that display the results in the notebook cell.



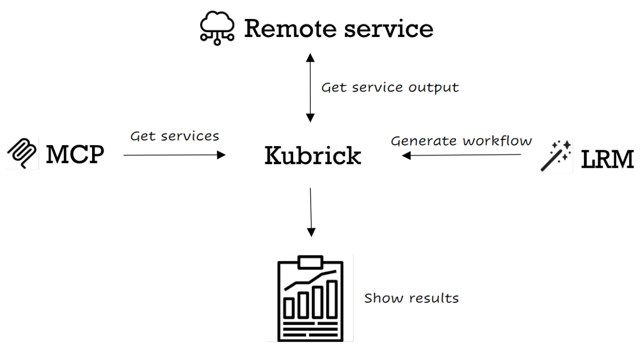
COMPILE-TIME WORKFLOWS

The glamoured *"agentic AI"* trend put LRMs in the main stage of complex programming exploiting their ability to elaborate decisions based on intricate set of input data. In the reasoning and acting (ReAct) paradigm the LLM is called during the transition in a broader state-machine that describes the the entire process. Basically, the AI tool is adopted not for planning the entire workflow where specialized modules can be engaged for specific tasks, but for deciding which action to take at each step of the process. It is a quite limiting approach that just raises the complexity of the overall solution.

Why not let the AI tools to generate the entire workflow at once? Attempts in this direction have demonstrated that it is possible to substantially improve the effectiveness of AI generators. This achievement reinforces the thesis that a highly expressive and declarative language like the one presented in this project can improve the impact of agentic AI.

MCP INTEGRATION

We can inform the generator about available functions and their interfaces (following Kubrick's language) and run the GenAI as an orchestrator and model the entire process in *compile-time*. Available functions can be those imported from libraries but also those exposed through model context protocols (MCP), a revisited protocol for service discovery. In this way, the single components can be invoked by the symbolic runtime that execute the generated code with more fine-grained control over its execution.


Remote service
Get service output
MCP
Get services
Kubrick
Generate workflow
LRM
Show results