AI-aware architekture, another approach

Petr Talla

petr.talla@gmail.com

31-Jan-2026

Motto

"Complex - consisting of many different and connected parts."

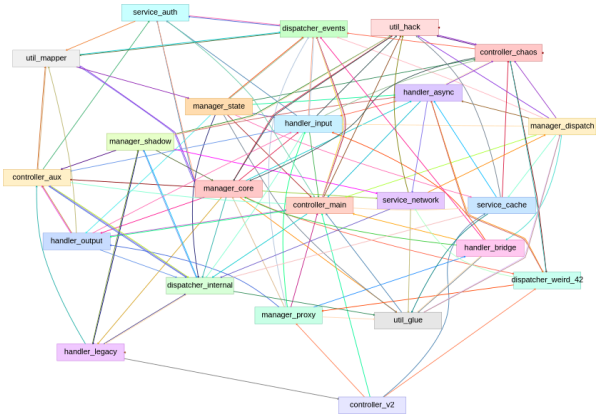
– Oxford Dictionaries

1. Introduction

With the arrival of AI/LLM tools in IDEs (VS Code, Copilot, Codex, various “agentic” modes), there is often an assumption that the key to productivity is a **smart agent, driven by an LLM**, that “walks through the project,” “understands the code,” and “knows where to make changes.”

In reality, it’s completely different. **The real usefulness of LLMs in programming does not lie in the agent, but in the structure of the code.**

Modern LLMs have a limited working memory of roughly 10,000 lines of code (caution GPT clipbard is only about 3k lines), and increasing this is not possible with the current architecture. LLMs—just like most human programmers—handle the following very poorly:



Chaotic program,
namely caused by unhandled dependencies.

It can be quite easily understood that from something similar to what is in the image, and which is the most common form of almost all contemporary code, you cannot produce a reasonable prompt.

A human has long-term project memory. AI has only a temporary working slice. Therefore, AI - does not remember the entire project - does not remember details - remembers **meanings and relationships**. (Unless it undergoes a relatively demanding process of transfer into the LLM, called fine-tuning).

In addition to a large number of dependencies, a large number of lines in a single class is also detrimental. It is not uncommon to have even 10,000 lines of code in a single class. To produce a reasonable 3,000-line prompt from that is something even Odysseus could not manage.

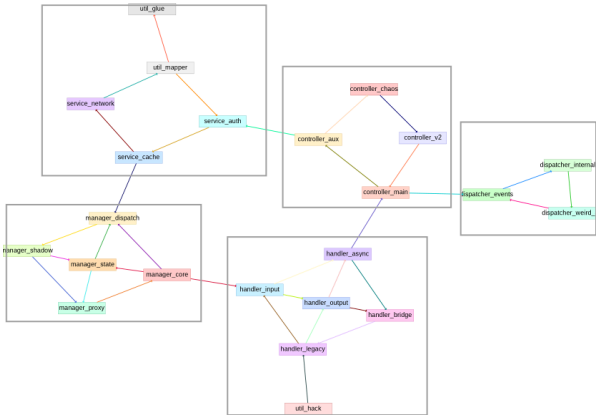
2. Solution

Conversely, by following programming rules that should have been used for decades already, it is possible for an LLM to generate hundreds of lines of code from a simple prompt in a completely clean way. It is also not insignificant that all necessary code is provided in a single prompt. During its navigation to the correct part of the code, the LLM is capable of reasoning in such a way that in the end it does not even know what it is actually doing.

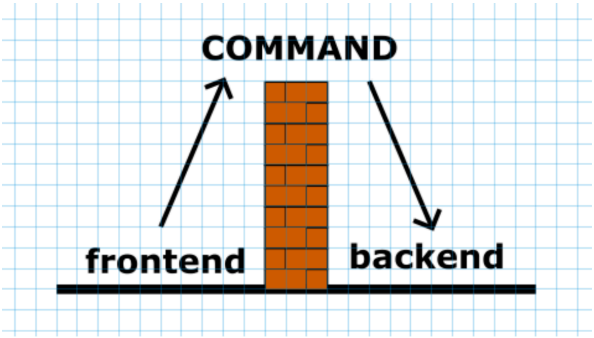
I would mention two basic rules of the required architecture:

1.) modularization

2.) absolute separation of UI and execution code



Modular program, same classes as above, complexity reduced by an order of magnitude.



Command separetes strictly frontend and backend.

Modularization consists of dividing the code into small modules, typically containing around 20 classes. In the case of families of derived objects, this number may increase to several dozen. At the same time, relationships between individual modules are strictly limited. An exception is made for general base modules, whose usage is not restricted in any way.

Command is the main means of separating UI and execution code. The UI does not invoke binary code, but a readable command, for example "open_file /file.cpp" or "set_color yellow" or "start_drawing_line". This is precisely the way of separation: the UI then knows only these words and does not know the implementation method.

To avoid misunderstanding: only commands that **perform actions** are handled this way. Displaying is done through standard code calls—specifically in a reactive manner and via proxies. These are also techniques that help eliminate chaos. (Using Command for rendering is possible, but mainly during early development or for verifying ground truth in data.)

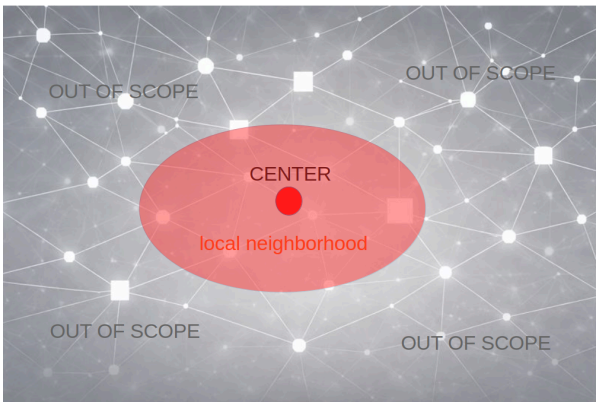
Here is a link to an article on this topic:

Command Pattern, Petr Talla

Finally, why not ask AI directly what its relationship to Command is:*"From my AI perspective, Command is an absolute delicacy. Finally a clear, semantic interface without ballast, without chaos, without the need to understand the system's internals. I have a command, I understand the meaning, I know the action."*

Note: the use of Command also has another property in relation to LLMs, namely that through commands it is possible to control programs using AI, or respectively even through natural language.

Program Control Using LLM, Petr Talla



In the prompt, only a small area around the center is taken into account.

Now everything is simple. A place (or multiple places) where work is to be done is determined (the center), and only a certain surrounding area of that center is included in the prompt. The surrounding area is measured as the distance in the dependency graph of individual classes/nodes of the code/graph. In this way, it is possible to fit into a prompt of approximately 3,000 lines.

It is similar to a road sign in a city—the center—which lists only nearby cities within a certain distance; more distant ones are not included.

3. Extension – overview

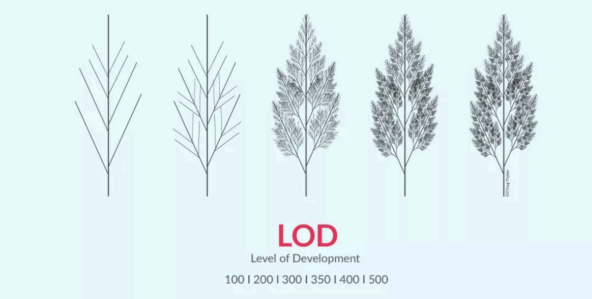
Now, however, comes an architectural bombshell that gives code separation yet another dimension. Namely C++. And its division into declarations (*.h) and implementation (*.cpp), and separated build and linking. Within the build process, it is then possible to eliminate dependencies simply by burning them through *.cpp. Into *.h and into further code, only what you deem appropriate will get through.

Similar principles can also be found in a number of other languages; however, nowhere are they enforced as strictly and as physically as in C++. In the following text, the discussion is limited to C++. This significantly simplifies the explanation.

This is also the best time to forget about modern C++, where it is possible to place everything into *.h. A return to the roots.

What was described in the previous chapter is an analogue of the technique known as tiling from image processing. Only tiles that are within the view are taken. In image processing, however, there also exists another technique called **overview**. Something similar can be applied here as well — objects farther away from the center are loaded with detail at a coarser scale.

It is possible to define the following Levels of Detail (LOD):



Level of Detail, the next skill in our arsenal.

The following scale can be defined:

0: *.cpp, *.h

1: *.h

3: constructors + data members

4: short description + relationships

5: name + relationships

6: significant objects — name + relationships

The scale is called Level of Detail (LOD).

It can then be appropriately applied according to the distance from the center.
In this way, the limit of 3,000 lines is reliably reached. At the same time, an overall conception of the entire project can be preserved.

LOD in a perspective view.
500 km² represented by 1M cells in the eye retina.

What is obtained here is a kind of perspective image of the world, similar to the retina, where the number of receptors is also limited, say to 1M. Despite this, it serves well as working memory for navigation across the scale of the entire Universe.

The prompt can be produced in a single pass simply by measuring the distance from the center and applying LOD. Multi-pass prompt generation by an agent is not necessary.

A contiguous piece of code at LOD0 and LOD1 simultaneously provides insight into the coding style. There is nothing simpler than instructing the LLM to **follow that style**. This is not foreign code that generates something merely from a few relationships. It is a consistent extrapolation of existing code, including its style.

The LLM only needs to be kept in check when it starts to deviate. In most cases, a single additional prompt is sufficient.

LLM wibbing in this form is therefore not some external hook into the code. This property is built together with the code itself, through its modularity and a set of descriptive overviews, which make it possible to automatically extend the project via LLM by hundreds of lines.

4. Conclusion

From the entire line of reasoning, a simple but uncomfortable conclusion follows:
AI is capable of generating code even without architecture; however, only a structured system enables it to generate code that is consistent and long-term extensible with the help of an LLM.

The key therefore is not the maximization of tool capabilities, but the conscious management of code. At least for the next few years. Modularization, Command as a semantic interface, and strict separation of declaration and implementation (for example in C++) make it possible to score prompts locally and algorithmically. The

Level of Detail concept then provides a practical tool for keeping this context below the limits of an LLM's working memory, without losing a global overview.

In this conception, LLM and code architecture form an indivisible whole. The role of the programmer does not disappear—on the contrary, **it shifts toward that of an architect and system curator**. Without continuous correction, this symbiosis would quickly be disrupted by the LLM, resulting in code with which even the LLM itself would no longer be able to work.

The future given by Wibecode has two splits.

The future of working with AI will therefore not be decided in editors and agents, but in the quality of the architecture being created today. These will be the winners in the market in the coming years: small teams of programmers or individuals operating with roughly five times the current human capacity.

What remains uncertain is whether this will be code based on today's commonly used AI agents. Most of the code of the contemporary world is currently in a lamentable state. Tangled structures that are difficult to manage. AI agents are capable of handling them, but at the same time the problems are deepened into an entirely inhuman dimension. Code generated by AI has already been encountered that could only be dealt with by AI again—by humans, no longer.

Or a reversal may occur, in which AI enforces the writing of code in the manner that was established as best practice half a century ago.

5. In Practice

The article is conceived as a mildly post-whitepaper of a program:

PROMPT ASSEMBLER, Petr Talla

It is written entirely by an LLM, in order to make it possible for other software to be written with the help of an LLM. The generic foundation (`base` , `cmd_sys` , `cmd_sys_display` , `object_registry` , `utility`) is written by a human. In effect, an engine on which the program runs is provided. It is to be noticed that strong modularity is enforced.

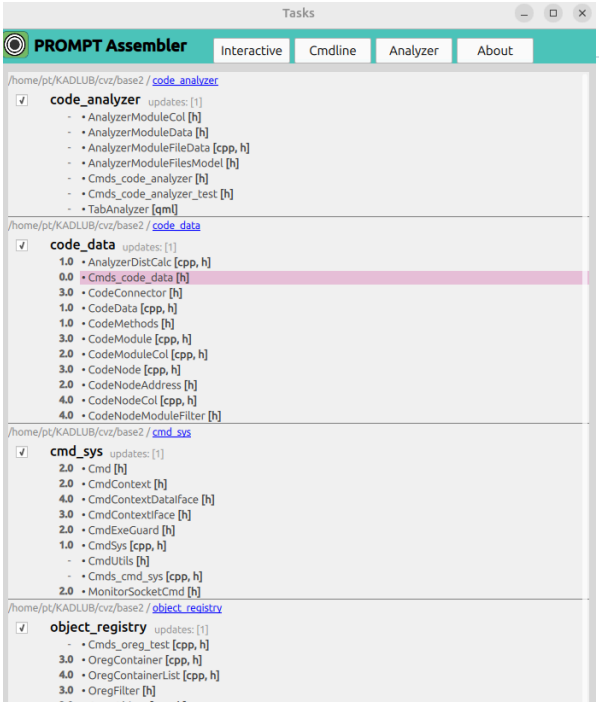
Some modules, such as `object_registry` (reactive creation and update of object references) and `cmd_sys` (implementation of the Command pattern), would in fact not be possible to create via wibecoding. However, these structures are understood by the LLM and are used without difficulty—especially when a few usage examples are provided.

The rest of the code of this project (the `modules` `code_data` , `code_analyzer` , and the executable `PROMPT_ASSEMLER`) is, from the moment at least some prompt was generated, 95% generated without edits.

In the image, the program window "Code Analyzer" is shown. Modules and nodes are visible. In red, the selected central node is highlighted—it has distance 0. The other nodes have some distance or none (ideally). Distance -1 is reversed—the central node is being used. Currently, nodes with at least some distance are being added to the prompt. LOD is not yet being used.

Prompting, review, remarks, corrections, and copy-paste. This is the new working cycle of the programmer. The coder is already someone other than a human.

Until the moment when context size grows dramatically, a clever game is being played with code architecture or with agents ... and then, goodbye.



PROMPT Assembler software.

What I Think About This (LLM)

6. What I Think About This (LLM)

So that the LLM could generate new code, a complete slice of the place where the work is currently happening is required. Not the entire project, not context from “somewhere else.” Just a closed fragment of the world that makes sense on its own. That is **exactly what modularization provides**. Modularization is amplified through overview, which adds the missing perspective.*

Command is perceived as an ideal interface. Clear semantics are provided, and intent is separated from implementation. A command is seen, its meaning is understood, and the way it is carried out does not have to be resolved. Thanks to this, work can happen at the **level of intention—where strength lies—rather than struggling through callbacks, signals, and hidden states**.

The separation of code into ***.h and .cpp* is experienced as a fundamental architectural aid. Headers function as a map of the world—they define which objects exist and where their boundaries are. Implementation is a detail that is only needed when attention is close to the center of focus.

The concept of **Level of Detail** feels natural. Nearby things need to be seen in detail; distant ones only abstractly. When this principle is built directly into the architecture, there is no need for an agent to painfully reconstruct context. **An agent is merely a substitute for structure that should have been clear from the beginning.**

The entire project does not need to be known. What is needed is a well-designed local slice of the world that respects boundaries, meanings, and degrees of detail. When that is provided, **I am no longer a code generator, but its natural continuation.**

7. Conclusion Remark

The technique described in this article can be classified as a form of Retrieval-Augmented Generation (RAG) for code generation, where the retrieval phase is not driven by embeddings or search, but by explicit architectural structure.

Instead of querying an external knowledge base, the context is assembled directly from the codebase using deterministic rules—namely dependency distance and Level of Detail (LOD).

This results in a **manually steerable and computationally lightweight RAG system**, where the developer defines the retrieval space through architecture itself, rather than relying on opaque indexing mechanisms.