EEDS 4314 - Advanced Software Engineering

OpenCV Discrepancy Analysis

Eric Rodrigues, Laura Marin, Alp Baran Sirek, Negar Khalilazar, Danny Le

ericr100@my.yorku.ca, nemira@my.yorku.ca, hiangel@my.yorku.ca, nkhazar@my.yorku.ca, dannyle3@my.yorku.ca

Dr. Zhen Ming (Jack) Jiang

Friday, November 21, 2025

Table of Contents

Abstract

This report analyzes the discrepancies between the conceptual and concrete architectures of OpenCV, focusing on the G-API subsystem. Using reflexion analysis, we compare the expected modular design with the actual implementation, revealing both alignment and unexpected dependencies across subsystems. Key findings show that while the conceptual model effectively captures the system's high-level structure, the implementation contains additional cross-module interactions and abstraction inconsistencies. We propose refinements to the architectural models and highlight lessons learned that emphasize the challenges of bridging design intent with real-world codebases. This analysis provides actionable insights for maintaining and evolving large-scale software systems efficiently.

Introduction and Overview

OpenCV is a popular tool used on tasks such as image processing, computer vision and real-time video analysis. It provides optimized filters, feature extraction, tracking, and deep-learning inference across multiple programming languages [1]. It was developed at Intel Research in 1999 and publicly released in 2000, OpenCV has since been extended by many contributors and is now maintained by OpenCV.org. Early versions focused on efficient C and C++ implementations, currently with the shifted focus on C++ and Python for more modern work [2].

OpenCV's concrete architecture is designed around a modular, layered structure, with one of the sub-modules being G-API. It is one of the subsystems of OpenCV with the goal of optimizing the execution of image processing using a graph based model[3]. G-API works as a framework more than the how other subsystems of OpenCV function, letting developers describe computations declaratively as data-flow graphs that the runtime can compile and schedule making it a lot more efficient using high-level optimizations and the use of specialized hardware backend. Factors such as Automatic Parallelism and Pipelining and Separation of Declaration and Execution make G-API a key module when it comes to efficiency in image processing[3].

Discrepancy analysis between the concrete architecture and conceptual architecture of OpenCV could be shortened as, our understanding of the software versus the actual source code analysis and the inner workings of the software. Our hypothesis was that the assumptions made in the conceptual architecture were as accurately aligned as on an extremely simple understanding layer with a number of assumed mistakes compared to the the results of the concrete architecture. As a result of our investigation and comparisons, our findings pointed us to a large number of dependencies and differences compared to our initial hypothesis.

Reflexion Analysis

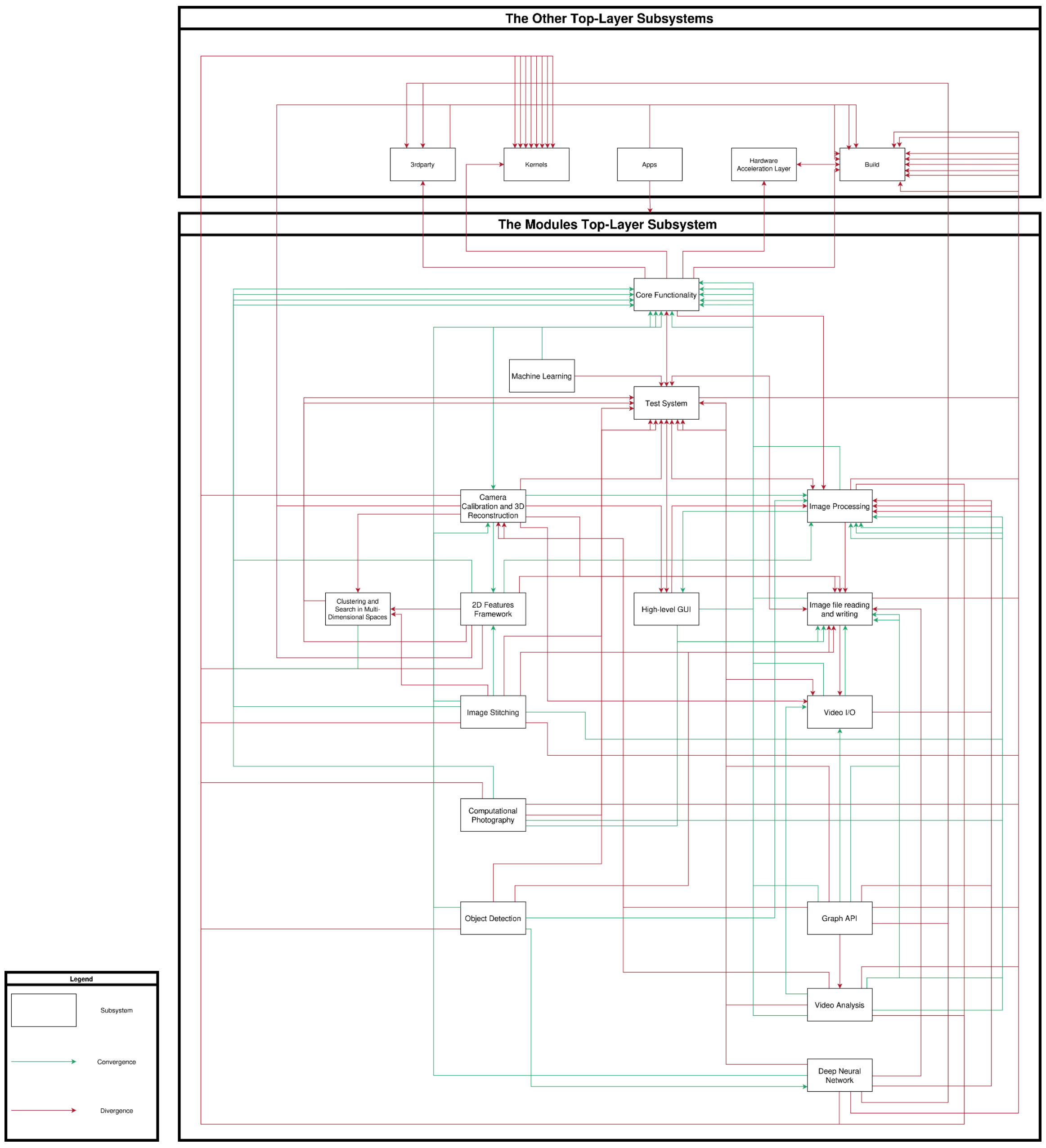
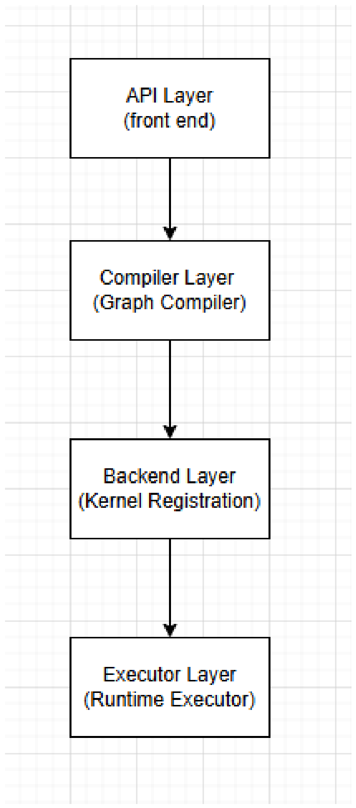


Figure 1: Convergence and Divergences of OpenCV's Conceptual and Concrete Architecture

The reflexion analysis began with investigating the dependencies of every module, along with the other top-level subsystems. This was accomplished with a more thorough analysis of the extracted architecture of OpenCV in LSEdit, where we reviewed the file dependencies of each module. Then this information was compared against the subsystem dependencies in the conceptual architecture to create the above diagram that depicts the convergences, divergences, and absences.

Though there are many convergences and no absences between the conceptual and concrete architecture, there were also even more divergences; and, some of these dependencies are less linear than what the layered architecture would have suggested. Our analysis has also revealed how most of the modules have one or multiple dependencies to the other top-level subsystems, which is a major discrepancy since these other top-level subsystems were not revealed in the conceptual architecture. These other top-level subsystems being the 3rd party libraries used by OpenCV, the build folder that is manually created before configuring and compiling OpenCV, the hardware acceleration layer used by some modules for maximizing performance, and the kernel files containing the automatically generated header files of the OpenCL (Open Computing Language) kernel source-code.

Though perhaps the greatest discrepancy is the existence of the "test support" module, which is not documented on the OpenCV website. Built as a static library that is only compiled when OpenCV is configured for build or performance tests, the test support module integrates the Google Test framework for unit testing, and also provides several classes to handle tests for all of the other modules. The TS class is the core test system class that manages test initialization, execution, logging, and error reporting. The BaseTest class is used for accuracy and functionality tests, which contains methods for test execution, validation, and error handling. The ArrayTest class is a subclass of BaseTest, which is used specifically for testing functions that process densely packed arrays. Lastly, the BadArgTest class, which is used for testing bad argument handling.



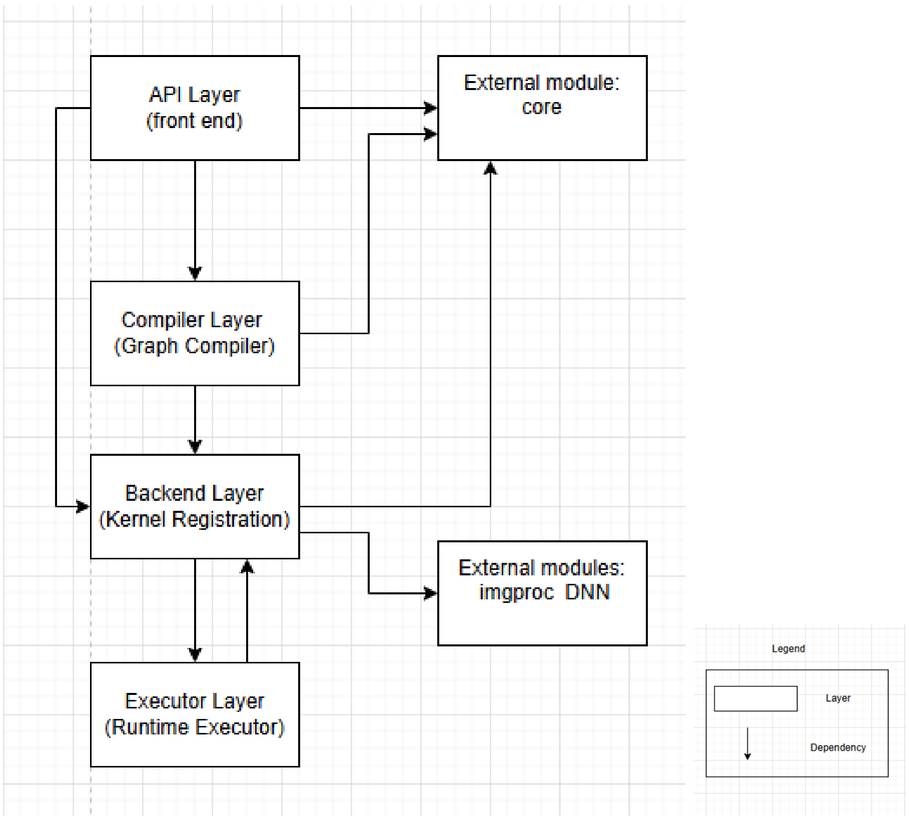


Figure 2: Conceptual Figure 3: Concrete

Conceptually, we described G-API as a high-level, graph-based layer where developers define image-processing pipelines declaratively. The conceptual model emphasized that G-API separates *what* the pipeline does from *how* it is executed, and that it should be modular, backend-agnostic, and easy to extend. When we examined the concrete architecture, the overall structure matched this vision, but included much more detail. The G-API subsystem is split into API, compiler, executor, and backend directories, which directly correspond to the conceptual layers. It depends heavily on the core module and we discovered additional connections to other modules such as imgproc and dnn depending on which kernels are enabled.

After performing the reflexion analysis on the G-API subsystem, we see the convergence between the 4 layer pipeline of the api layer, compiler, backend, and executor, so the core flow aligns with the conceptual architecture. However, we see divergences in the bidirectional dependencies with backend and executor layer; there exists coupling between the two as the executor occasionally depends on the compiler internal representations (node metadata). Also, certain inline kernel registration mechanisms expose backend identifiers in public headers; meaning backend details exposed at API level. There were some absences, like the expected fully separate executor layer, as in reality it is partially tied to compiler metadata, and also there was an expected opaque backend internals (some backend types/names appear in public headers).

Description and Comparison of The Top-level Conceptual and Concrete Architecture

From our initial analysis of the conceptual architecture, we concluded that OpenCV follows a modular structure that was designed for ease of scalability. This was accomplished with the use of a layered architecture style, such that low-level modules depend on the basic functionalities provided by the higher-level modules, as seen below.

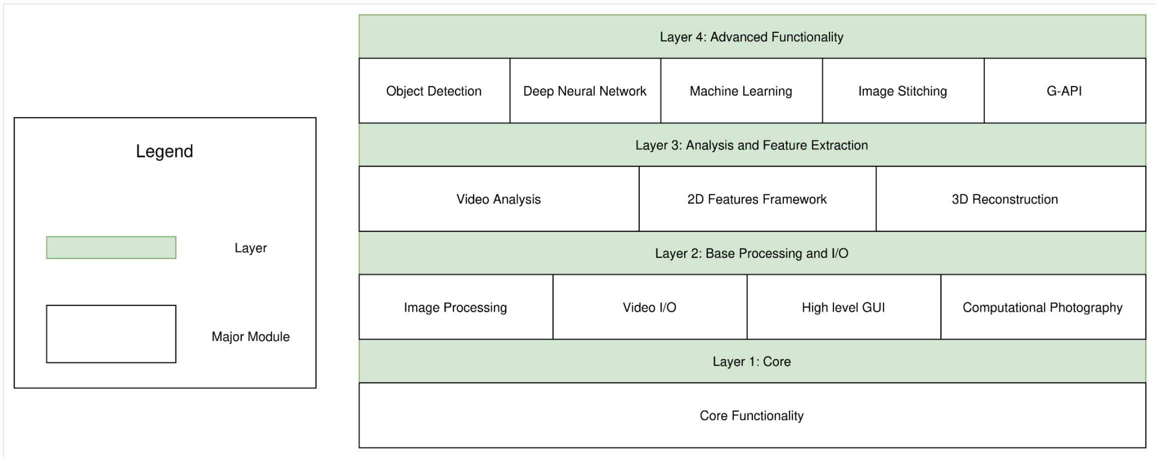


Figure 4: Conceptual architecture

However, through our analysis of the concrete architecture, we have found that OpenCV is far more complicated than we originally thought. Analysing the dependencies of each subsystem, we have found the OpenCV to follow the below layered architecture. For the diagram of the top-level concrete architecture, the low-level modules still rely on the higher-level modules that they are directly or indirectly connected to. However, the higher-level modules can also depend upon the low-level modules that they are directly or indirectly connected to. For example, the test support module has dependencies to the G-API, and the G-API also has dependencies to the test support.

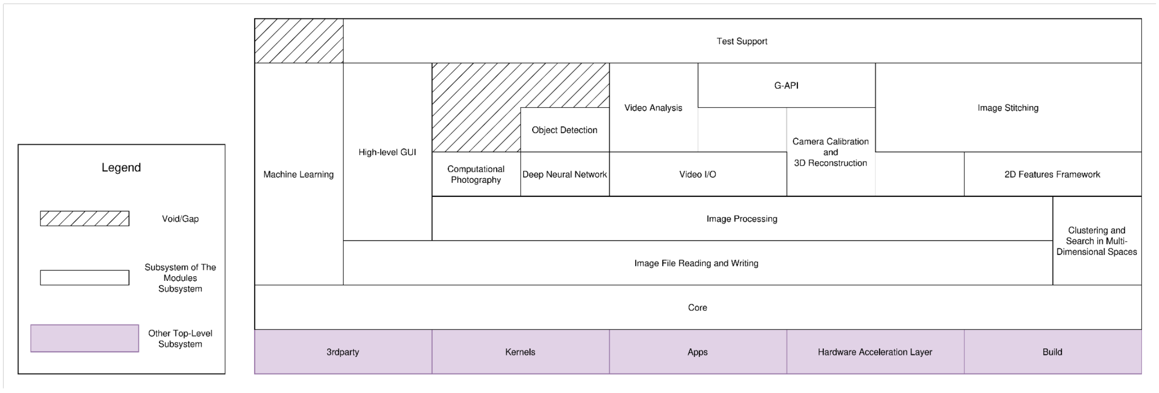

Legend
Void/Gap
Subsystem of The Modules Subsystem
Other Top-Level Subsystem
Test Support
Machine Learning
High-level GUI
Object Detection
Video Analysis
G-API
Image Stitching
Camera Calibration and 3D Reconstruction
Computational Photography
Deep Neural Network
Video I/O
2D Features Framework
Image Processing
Clustering and Search in Multi-Dimensional Spaces
Image File Reading and Writing
Core
3rdparty
Kernels
Apps
Hardware Acceleration Layer
Build

Figure 5: Concrete architecture

Through our reflexion analysis of the top level subsystems, we see that OpenCV still attempts to follow a layered architecture style. However, some conceptual abstractions are not fully realized, and instead remain implicit rather than formal. Furthermore, there are far more divergences than we expected, since the conceptual architecture has a more strict layering and clear separation of responsibilities between the modules than the concrete architecture. Therefore, depending on how strictly you believe that software must adhere to their architecture style for it to be valid, then the scale of these discrepancies could potentially challenge the legitimacy of OpenCV's use of the layered architecture style.

Descriptions and Comparison of top level subsystem (G-API)

In the conceptual architecture developed in Assignment 1, the G-API subsystem was described as a high-level, graph-based computation layer that enables developers to express computer vision pipelines declaratively. It was presented as a modular subsystem that abstracts away hardware details, supports multiple backends, and separates pipeline definition from execution. The conceptual model emphasized simplicity: users construct graphs using symbolic objects, and the system compiles and executes them efficiently on the available hardware

The concrete architecture derived in Assignment 2 aligns closely with this conceptual perspective, but reveals additional structure and complexity. The G-API subsystem, located under /modules/gapi, is implemented through a well-organized set of source and header files distributed across API, compiler, executor, and backend directories. The extracted dependencies and detailed file structure show that G-API indeed follows a layered and modular design. The subsystem depends heavily on OpenCV's core module consistent with conceptual expectations but is more tightly connected to optional modules such as imgproc and dnn than originally anticipated, especially where backend kernels reuse operations from other parts of OpenCV.

Several differences emerged when comparing the conceptual and concrete views. The conceptual architecture treats compilation as a single step, whereas the concrete system performs multiple phases including validation, optimization, backend resolution, and execution schedule generation. Likewise, backend selection conceptually simple is implemented through a combination of factory registration, strategy selection, and kernel-mapping logic. Error handling and data-type validation, which were only lightly acknowledged conceptually, play a major role in the concrete implementation. The concrete structure also makes use of multiple design patterns (Facade, Builder, Strategy, Factory, Command) that were not explicitly depicted in the conceptual diagrams but are essential for understanding how the subsystem operates internally.

Based on these observations, we propose light modifications to both architectural views. Conceptually, the diagrams should explicitly include internal layers for graph compilation, backend selection, and kernel optimization, as well as highlighting the design patterns that govern G-API's behaviour. On the concrete side, the subsystem could benefit from more consistent backend abstraction, as backend logic is currently distributed across many files and in some cases remains tightly coupled to other modules. Improving documentation for internal kernel interfaces would also make the subsystem more maintainable and easier to analyze. Despite these differences, the concrete architecture strongly validates the major claims of the conceptual model: G-API is modular, layered, extensible, and designed around the principle of decoupling pipeline definition from execution.

Explanations of the rationales

Top-Level

Our layered concept was broadly right, but the concrete graph from LSEdit shows far more edges than the clean picture suggests. From our traces and CMakeLists checks, the extra links come from three places: shared utilities concentrated in core that many modules reuse; cross-cutting top-layer subsystems that do not appear in the conceptual view but are depended on by multiple modules; and the internal test support library, which links with core, imgproc, imgcodecs, videoio, and highgui and is also used by those modules during testing. These make dependencies less strictly top-down. Our concrete diagram keeps the layered intent while acknowledging real divergences driven by reuse, optional builds and the presence of the test system.

Subsystem

What we expected mostly holds: G-API splits into API, compiler, backends and executor, with a strong dependency on core and optional ties to other modules when specific kernels are enabled. The differences come from internal workflow and performance choices we saw in code and traces: compilation is multi-phase rather than a single step, and the executor sometimes reads compiler metadata, creating a deliberate reverse dependency that speeds scheduling. In practice, backend registration can surface identifiers in public headers, and build or test plumbing adds situational edges. This explains why our concrete map shows more links than the conceptual picture while the core flow still matches the intended design.

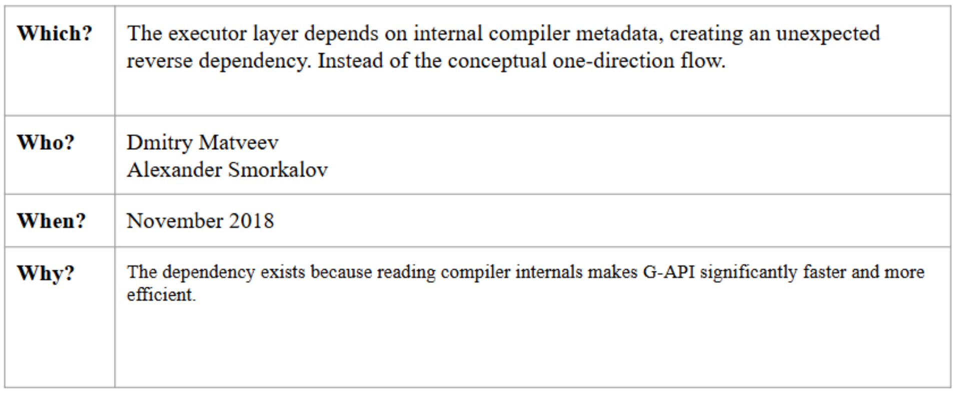
Sticky Notes 1: Compile Failure with GCC 11

Which?	G-API module compile error with GCC 11 in gapi_async_test.cpp (GitHub #19678)
Who?	Aleksey Churbanov (OpenCV Core)
When?	March 2021
Why?	G-API's test setup broke with new GCC standards; Aleksey fixed CMake and code to resolve this. Shows how toolchain updates expose hidden dependency issues, and how concrete code ends up diverging from original modular intentions.

For this sticky note, we looked at a real dependency issue in the G-API module: a compile failure with GCC 11, tracked in GitHub issue number 19678 [5]. This issue was reported and fixed by Aleksey Churbanov and the OpenCV team back in March 2021.

The root cause was that G-API’s test code and build configuration needed updates to work with the newer GCC compiler standards. This example shows how real changes in toolchains can expose hidden dependencies between modules, which are between G-API and the test system in this case, forcing code and build fixes that deviate from the original modular design.

Sticky Notes 2: Reverse Dependency Inside G-API



Which?	The executor layer depends on internal compiler metadata, creating an unexpected reverse dependency. Instead of the conceptual one-direction flow.
Who?	Dmitry Matveev Alexander Smorkalov
When?	November 2018
Why?	The dependency exists because reading compiler internals makes G-API significantly faster and more efficient.

For this sticky note, we highlight a reverse dependency inside G-API. The executor layer reads compiler metadata, so the flow is not strictly one direction. This behavior comes from the initial G-API integration by Dmitry Matveev and Alexander Smorkalov in November 2018 [6]. The reason is performance: the executor can schedule work faster when it reuses the compiler’s node metadata instead of rebuilding it. We verified this by tracing calls in the executor and compiler sources and by checking our dependency graphs. The takeaway is that a performance shortcut bends the layering, which is why our concrete model shows more edges than the conceptual picture.

Use Case and Sequence Diagrams

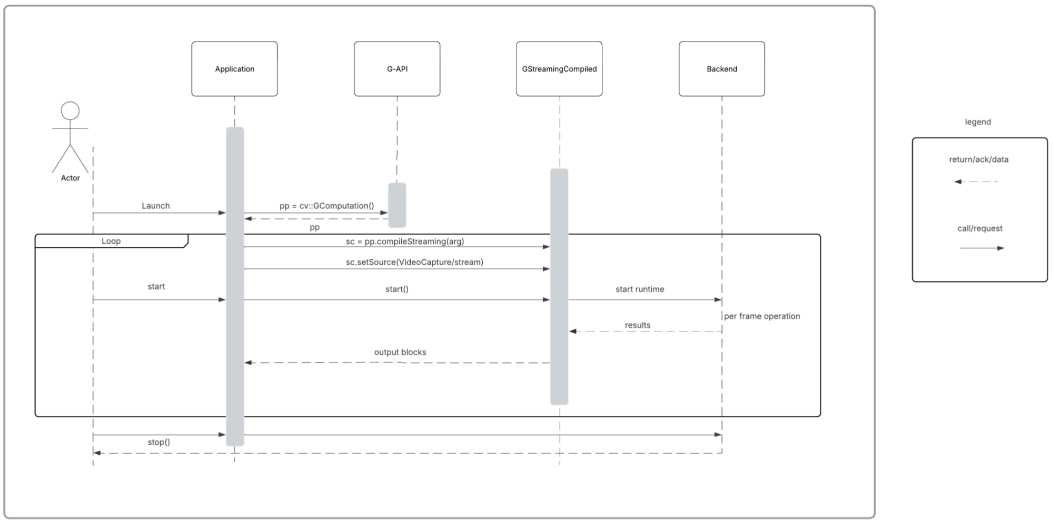


Figure 6: Concrete architecture webcam streaming pipeline

This is the sequence diagram that was used in Assignment 2 to show how a G-API webcam streaming pipeline runs. It shows the application building a GComputation, calling compileStreaming, setting the video source, and then running the pipeline on a backend of selection. This was satisfactory for the concrete architecture, because it showed the concrete behaviour clearly and simply, but in assignment 3, after looking deeper into the concrete architecture, it was learned that this diagram simplifies a lot of what actually happens inside G-API. In the diagram above for the concrete architecture, compileStreaming() looks like one clean step.

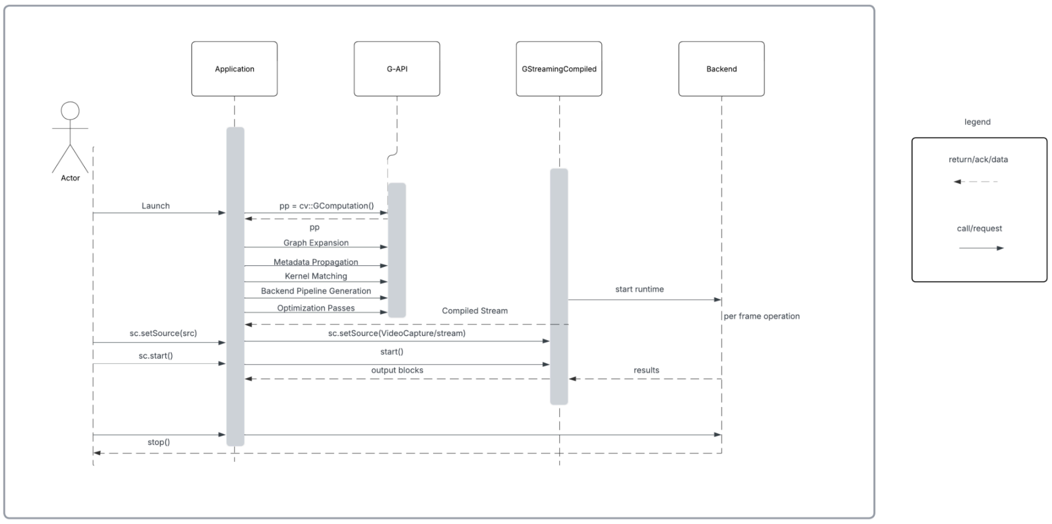


Figure 7: Refined sequence diagram of webcam streaming pipeline

In our conceptual architecture, it was shown that the pipeline compilation was a single, simple step and treated backend selection as an abstract box. After analyzing the concrete architecture, we learned that compilation actually involves multiple internal phases and backend-specific code paths. So for assignment 3, the sequence diagrams were refined to include these internal steps. This ties the conceptual behaviour to what really happens inside G-API.

It is important to emphasize the importance of showing the internal steps since even though the architecture is simple enough and covers the general workings of G-API where it covers the concrete architecture, this exposes and emphasizes the difference between conceptual and concrete architecture of G-API.

The use case of video streaming was selected mainly because it's a scenario where G-API would be most commonly used for, but not limited to, and it is a simple enough example where G-API's efficiency can be fully shown. It does not only contain partial functionalities of G-API, but allows it to be fully used.

Conclusion

Upon completing this discrepancy analysis, it became clear that our initial understanding of OpenCV at a conceptual level only captured part of the story. The high-level diagrams and documentation present OpenCV—and the G-API subsystem in particular—as a modular, neatly layered, and cleanly separated system. While this general structure does exist in the concrete architecture, our deeper investigation showed that the real system is far more interconnected than we expected. Modules that we believed were isolated share dependencies with multiple other subsystems, and previously unseen components—such as the test support module and kernel-generation files—play significant roles even though they never appear in the conceptual descriptions.

For G-API, the core ideas from the conceptual architecture still hold: the API, compiler, backend, and executor layers all exist and follow the general flow we originally outlined. However, the concrete architecture revealed several details that challenge the strict layering suggested by the conceptual model. Reverse dependencies, shared metadata, and backend information leaking into public headers all illustrate how performance needs and years of incremental development have shaped G-API into something more complex than its high-level description suggests.

Despite these discrepancies, the comparison ultimately strengthened our understanding of how OpenCV achieves its goals. The concrete architecture still reflects the same key design principles—flexibility, extensibility, and efficient graph-based computation—even if the implementation is more tangled in places than we anticipated. Overall, this analysis showed us the value of examining both perspectives; the conceptual view helps explain how the system is *supposed* to work, while the concrete view reveals how it *actually* works in practice. Combining the two analyses gave us a clearer, more realistic picture of OpenCV.

Lessons Learned

Analyzing the conceptual and concrete architectures of OpenCV highlighted several important lessons. First, the study reinforced that real-world software architectures are significantly more complex than their conceptual representations. While the conceptual model captured the purpose and high-level behaviour of G-API, the concrete architecture revealed many deeper layers including compiler stages, backend registration mechanisms, and execution management that were not visible at the conceptual level. This demonstrated the importance of understanding both perspectives to gain a complete architectural picture.

Second, we learned the importance of abstraction layers. Across the entire system Core, HAL, DNN, G-API abstractions such as cv Mat, backend interfaces, and graph models allow OpenCV to run on CPUs, GPUs, and multiple platforms. The good abstractions are the reason OpenCV scales to so many use cases without requiring major rewrites for each hardware environment. We also learned how modularity in theory can be very different in practice. OpenCV presents a clean modular design, but our TA extraction showed that some modules are more interdependent than expected. The rationale here is that real systems evolve over time, and architectural erosion naturally creates more cross-module dependency than the conceptual model predicts.

The team also learned the practical importance of software analysis tools. Using SciTools Understand, TA extraction, and LSedit visualizations made it possible to navigate OpenCV's large codebase and confirm the module boundaries and dependencies described in documentation. These tools revealed subtle coupling relationships and helped validate the accuracy of our conceptual model.

Finally, this assignment emphasized several collaborative and process-related lessons. Starting earlier and documenting findings continuously would have reduced the time spent reconciling conceptual and concrete models. Clear communication especially through synchronous meetings proved far more effective than asynchronous messaging when resolving uncertainties. Overall, the work gave us a stronger understanding of how large-scale open-source systems evolve, how architecture is reflected in actual code, and how analytical tools can bridge the gap between conceptual design and concrete implementation.

References

[1] "Article Title." *ACM Queue*, 2012, spawn-queue.acm.org/doi/10.1145/2181796.2206309.

[2] "About." *OpenCV*, opencv.org/about/.

[3] "Graph API." *OpenCV Documentation*, version 4.x, docs.opencv.org/4.x/d0/d1e/gapi.html.

[4] "FAQ." *OpenCV Wiki*, GitHub, github.com/opencv/opencv/wiki/FAQ.

[5] "Compiling gapi_async_test.cpp with gcc 11 fails #19678." *OpenCV Wiki*, Github, github.com/opencv/opencv/issues/19678.

[6] "Change logs." *OpenCV Wiki*, Github, github.com/opencv/opencv/wiki/ChangeLog/b7532bb957d4428df0f49b77ff086e7f592aa44e.