EECS 4314 - Advanced Software Engineering

OpenCV Concrete Architecture

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

Monday, October 10, 2025

Table of Contents

Abstract

The purpose of this report is to present the concrete architecture of OpenCV with a focused analysis of its ***G-API*** subsystem. This report will focus on the derivation process of OpenCV's subsystems with a deeper dive into G-API as the studied software system. This report first outlines the derivation process used to obtain a concrete view of OpenCV's structure. Architectural entities and static dependencies are extracted using "SciTools Understand", exporting the dependencies with a ".ta" representation, visualising the results using LSEdit. To further analyse the subsystems, focus is given to G-API, outlining its role as a graph based computation layer and previewing its sub-directories and dependencies. Attention is directed to concurrency and performance considerations. The report concludes by a simple use case being provided, with a discussion of lessons learned during the analysis of the concrete architecture of OpenCV. By the end of this report, readers should have a clear understanding of OpenCV's concrete architecture and the internal structure of the G-API subsystem.

Introduction and Overview

OpenCV (the Open Source Computer Vision Library) is a widely used toolkit for 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]. 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 of computer-vision algorithms, with a strong emphasis on real-time performance.

OpenCV has integrated with popular machine learning and deep learning frameworks such as TensorFlow, PyTorch, and ONNX, enabling developers to combine traditional vision techniques with modern AI models. Recently, a key milestone in OpenCV's evolution has been the inclusion of GPU acceleration through CUDA and OpenCL. Before image processing in OpenCV was executed on the CPU, complex operations were computationally slow. By introducing GPU acceleration, OpenCV can now offload many of these operations to graphics processing units. Unlike CPUs, GPUs consist of thousands of smaller cores optimized for performing many operations in parallel. This parallelism makes them ideal for tasks where the same computation needs to be applied repeatedly across large datasets, such as applying filters to every pixel in an image or running deep learning inference across batches of images. This, along with many other features, has been implemented into OpenCV to cover a wide range of tasks, helping it continue to be one of the most widely used computer vision frameworks in both academics and industry.

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[1]. 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[9].

Architecture Derivation Process

The process of deriving OpenCV's architecture begins with downloading the source code from GitHub, and creating a "build" folder within OpenCV's main directory. Opening a Terminal window within the build folder (or command prompt on Windows), we executed the command "cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ../" to generate the JSON compilation database file "compile_commands.json". This JSON file contains the compilation parameters, such as include paths[2]. This is used by SciTools' analysis software "Understand" to more accurately analyze OpenCV's file dependencies. To use Understand, we created a new project that points to the root directory containing our local copy of OpenCV's source code. Once Understand's analysis was complete, we exported the file dependencies as "opencv-4_UnderstandFileDependencies.csv". We then used our modified perl script "transformUnderstand.pl" to parse the CSV file, which creates the raw Tuple-Attribute (TA) file "opencv-4_UnderstandFileDependencies.raw.ta"[3]. From here, our group wrote a script in Java that reads the TA file to create our custom containment file—which describes the hierarchy of OpenCV's sub-directories and files[4]. In essence, the Java program reads a line of the TA file, splits the file path by each sub-directory, and writes to the output file the hierarchical relationships between each successive level towards that file. The Java program also checks to avoid repeating the same relationship for subsequent files. This continues until the TA file has no more unread lines, at which point it will write to memory the containment file "opencv-4_UnderstandFileDependencies.contain". Then by running our modified script "createContainment.sh", we imposed our containment onto the extracted data[5]. We then ran our slightly modified shell script "runLSEdit.sh" to visualize the extracted system architecture of OpenCV in LSEdit[6]. Finally, our group analysed the concrete architecture of OpenCV's top-layer subsystems and their inter-dependencies[7].

The "3rdparty" subsystem contains bundled third-party libraries and third-party dependencies, which OpenCV can optionally build and use in its other subsystems. It relies on the "build" subsystem, as it will output compiled artifacts into specific locations within the "build" folder.

The "build" subsystem is manually created by developers before configuring and compiling OpenCV, serving to keep all build-related files and compilation outputs separate from the source code. During OpenCV's compilation, the "modules" and "hal" subsystems interact with the "build" subsystem to establish dependencies, generate artifacts, and more.

The "apps" subsystem contains utility applications that are built on top of the OpenCV library. These are standalone command-line tools that provide specific functionality for computer-vision tasks. The "apps" subsystem depends on the "build" subsystem to access the generated configuration headers that are created during compilation. The "apps" subsystem also depends on the "modules" subsystem, since it needs to include module headers, access paths from the modules' source directories, and more.

The "kernels" subsystem contains automatically generated header files of the OpenCL (Open Computing Language) kernel source-code, which stores GPU-accelerated kernel code directly in the compiled libraries. These kernel files are not found in the source code , but are generated if OpenCL support is enabled. There are no dependencies for this subsystem.

The “modules” subsystem contains the core subsystems, each representing a distinct area of functionality related to computer vision and machine learning tasks. We can see this by entering the subsystem, and can thus see all of OpenCV’s main modules.[8] Returning to the top-level, we can see that the “modules” subsystem depends on the “hal” subsystem to provide accelerated implementations of common computer-vision operations. This separation provides hardware-optimizations without changing the core module’s general-purpose code.

The “modules” subsystem also depends on the “3rdparty” subsystem to provide third-party libraries that OpenCV modules require for specific functionality, such as: image codecs, compression, math, and more. The “modules” subsystem requires the “build” subsystem for its generated configuration files, which the modules need during compilation. They also need the “kernel” subsystem to provide GPU-accelerated implementations.

The “hal” subsystem contains the “Hardware Acceleration Layer”, providing optimized algorithms for various hardware vendors and acceleration libraries. The “hal” subsystem depends on headers from the “modules” substem, in order to access OpenCV's data structures and interfaces. The “hal” subsystem also depends on the “build” subsystem, as it contains a generated header file that acts as a bridge between the “hal” subsystem and the “modules” subsystem.

Architecture

OpenCV's concrete architecture is built on a modular and layered design, implemented primarily in C++ under the modules/ directory. Each module compiles into a separate library that can be linked independently while maintaining consistent interfaces and shared data structures. Lower-level modules provide core functionality, while higher-level modules build upon them to enable more specialized operations such as deep learning, video processing, and graph-based computation. At the foundation of the system is the Core module, which defines OpenCV's essential data structures, including cv:Mat and cv:UMat, as well as memory management, error handling, and mathematical utilities. Nearly every other module depends on Core, as it provides the standard interface for representing image and matrix data across the library.

Above Core, the Imgproc module implements the fundamental image processing operations, such as filtering, geometric transformations, colour space conversions, and edge detection. It uses the data types defined in Core and forms the basis for most image manipulation tasks in OpenCV. The Video and VideoIO modules extend these capabilities to motion and streaming data. Video provides motion analysis and tracking algorithms, while VideoIO manages frame capture from cameras and file input/output. These modules depend on both Core and Imgproc and also interface with platform-dependent backends through abstract adapter classes. The DNN module adds a deep learning inference engine that supports TensorFlow, Caffe, and ONNX models. It uses Core for data handling and integrates with Imgproc for preprocessing tasks. The module is designed around a plugin-style backend system, allowing execution on various hardware targets such as CPU, CUDA, and OpenCL through dynamic registration of backend kernels. The G-API module introduces a graph-based computation framework that allows image processing tasks to be expressed as computation graphs. Internally, it is divided into components for graph compilation, backend execution, and kernel management. G-API interacts with nearly all other subsystems, using Core data structures and delegating operations to Imgproc, DNN, or other backends depending on the pipeline configuration. The HighGUI module provides user interface utilities for displaying images and videos, as well as handling keyboard and mouse input. It serves as a lightweight visualization layer built on top of Core and VideoIO, mainly for debugging and rapid prototyping. The ML module, on the other hand, offers classical machine learning algorithms like SVM, kNN, and decision trees, using Core's matrix structures for data representation.

All these modules interact through the common data abstractions defined in Core, ensuring consistent memory handling and type compatibility. High-level modules rarely depend directly on each other, instead communicating through shared data and APIs. This design promotes modularity, enabling OpenCV to be used flexibly across different applications and platforms from embedded systems to full-scale computer vision frameworks. In summary, OpenCV's concrete architecture follows a layered and modular pattern. The Core and Imgproc modules form the foundation, while higher-level components such as DNN, G-API, VideoIO, and ML build upon them to provide specialized functionality. This dependency structure allows efficient reuse of core capabilities while maintaining clear separation between layers.

G-API Subsystem

The Graph-API or G-API subsystem is located under /modules/gapi, and provides OpenCV's graph-based computation engine. Unlike traditional procedural modules such as imgproc or dnn, G-API represents image processing operations as computation graphs that can be compiled and executed on various backends. The module is made up of approximately 290 sources and header files distributed across several subdirectories, which include its source, compiler, backend, and include folders. Each of these directories covers a distinct aspect of the subsystem, from user-facing APIs to backend-specific implementations. The subsystem depends heavily on the core module and, in some cases, can rely heavily on other subsystems such as imgproc and dnn, depending on which kernels and backends are enabled.

Understanding each directory within the G-API subsystem is essential to grasping how it integrates into the broader OpenCV architecture. The /include/opencv2/gapi directory contains public headers that define user-facing classes such as GMat, GScalar, and GComputation, which form the interface layer for building computation graphs. The /src/compiler directory houses classes and utilities for transforming high-level computation graphs into executable pipelines, including GCompiler, GCompilerBackend, and various optimization components. Execution is managed within /src/executor, which implements GExecutor and GCompiled to handle graph scheduling and backend kernel invocation. The /src/backends directory contains backend-specific implementations such as Fluid, OpenCL, and CPU backends each registering its own kernel set via factory macros. The /src/gkernel directory defines the kernel abstraction layer, including GKernel, GBackend, and GKernelPackage, which facilitate backend registration and operation discovery. Finally, the /test directory includes functional and performance tests that verify the correctness and efficiency of graph compilation and backend execution.

G-API also features a lot of class-level dependencies; the following are internal interactions that occur within the subsystem.

1. Graph Construction (Front-End API)

The user defines a computation graph using instances of GMat, GScalar, and GComputation. These classes reside in /include/opencv2/gapi/gcomputation.hpp and related headers. No immediate computation occurs; instead, an internal graph representation (GModel) is built.

2. Compilation Phase (Graph Compiler)

The GComputation compile() method instantiates a GCompiler object, defined in /src/compiler/gcompiler.cpp. The compiler performs dependency analysis, topological sorting of nodes, and kernel mapping using backend registration information from /src/gkernel.

3. Backend Binding (Kernel Registration)

Each backend registers its operations using GBackend Kernels() methods. These registrations are handled through macros like GAPI_OCV_KERNEL and are stored in GKernelPackage containers.

4. Execution Phase (Runtime Executor)

The compiled graph is wrapped in a GCompiled object. At runtime, GExecutor (in /src/executor/gexecutor.cpp) executes the sequence of kernels, passing data between nodes represented by GArg and GRunArg structures. Data conversion between Mat and backend-specific types occurs within these execution routines.

The data flow in G-API revolves around the execution of a computational graph that defines image processing operations and their dependencies. When a user builds a G-API pipeline, each operation such as filtering, colour conversion, or edge detection is represented as a node in the graph, while the edges represent the flow of data between operations. Input images are first encapsulated into GMat objects, which are symbolic representations rather than concrete data. During compilation, G-API transforms this abstract graph into an optimized backend-specific execution plan. At runtime, the data flows through the compiled graph where actual Mat data replaces symbolic placeholders, and each backend node processes its portion of the data before passing results downstream. This design enables G-API to decouple pipeline definition from execution, allowing the same graph to be efficiently executed on CPUs, GPUs, or heterogeneous systems.

Within the G-API subsystem, several components interact closely to translate high-level graph definitions into efficient runtime execution. The Graph Compiler is responsible for analyzing the user-defined pipeline, applying optimizations, and mapping nodes to available backends such as OpenCL or CPU implementations. The Runtime Executor then orchestrates the execution of the compiled graph, invoking backend kernels in the proper order and managing memory buffers between stages. The Backend Interface serves as the bridge between the abstract graph and concrete hardware implementations, exposing standardized APIs that allow developers to extend G-API with custom operations. These interactions ensure a modular and flexible execution model, where each layer operates independently yet cooperatively to achieve high performance and portability.

G-API employs multiple mechanisms to ensure robustness and high performance during graph execution. Error handling is primarily managed through OpenCV's exception system, which propagates errors such as invalid graph construction, unsupported operations, or data type mismatches. During graph compilation, G-API performs validation checks to detect structural inconsistencies before runtime. On the performance side, G-API leverages graph-level optimizations, including operation fusion, lazy evaluation, and data reuse to minimize redundant computations and memory transfers. Additionally, backend selection allows performance tuning based on hardware capabilities. GPU backends can accelerate parallel operations, while CPU backends optimize for low-latency execution. This design ensures that G-API maintains both correctness and efficiency across diverse computing environments.

The G-API subsystem represents one of the more modern, modular components of OpenCV's architecture. Its code organization clearly separates API, compilation, and execution responsibilities. The extracted dependency relationships show that it is tightly coupled to the core module but remains loosely coupled with the rest of the system. Through its graph-based design and backend abstraction layer, G-API extends OpenCV's

architecture to support optimized, parallelized pipelines while maintaining a clean modular boundary in the concrete codebase.

External Interfaces

The architecture of the OpenCV G-API is designed to sit precisely at the boundary between the high-level applications that rely on the library and the low-level execution frameworks that the library engages for performance [9]. At the top interface, user code interacts with symbolic classes such as cv::GMat, cv::GScalar and cv::GComputation, thereby declaring what operations are to be performed rather than how they will be executed [10]. This design enables the pipeline to remain hardware-agnostic while still integrating tightly with OpenCV's internals [11].

When the pipeline is submitted for execution by invoking compile() or apply(), G-API transitions into its compiler layer where the graph structure composed of data nodes and operation nodes is analyzed, optimized and partitioned according to available back-ends [9]. In this layer dependencies are resolved, nodes are clustered, and each operation is mapped to a kernel implementation specific to a backend (for example CPU, Fluid or OpenCL) [12].

Control then flows downward to the backend layer, where the chosen kernels run on the actual execution target using the memory abstractions and scheduling policies appropriate to that device [9]. For example, if an OpenCL backend is selected then the intermediate data may be handled in cv::UMat form and remain resident on the accelerator rather than being copied back and forth to host memory [13].

In terms of data flow, G-API employs a directed acyclic graph structure reminiscent of a pipe-and-filter architecture [10]. Inputs are declared symbolically, then bound to concrete containers (such as cv::Mat or cv::UMat) at runtime and processed through the scheduled kernels in topological order [11]. Each kernel consumes data and produces results, which feed subsequent kernels, so that data flows from source to sink across the compiled graph [14].

In streaming scenarios, G-API further enhances performance by pipelining frames through the graph: while one stage is processing frame N, another stage may already be processing frame N+1, enabling concurrency and higher throughput without changing the user-written pipeline logic [15].

Thus, from the user's perspective the interface remains simple and high-level, i.e. describe the computation once and hand it off, while internally the system governs control and data movement across layered components and hardware architectures. This separation ensures that the application remains portable, the architecture remains modular, and the execution remains efficient.

Use Case and Diagrams

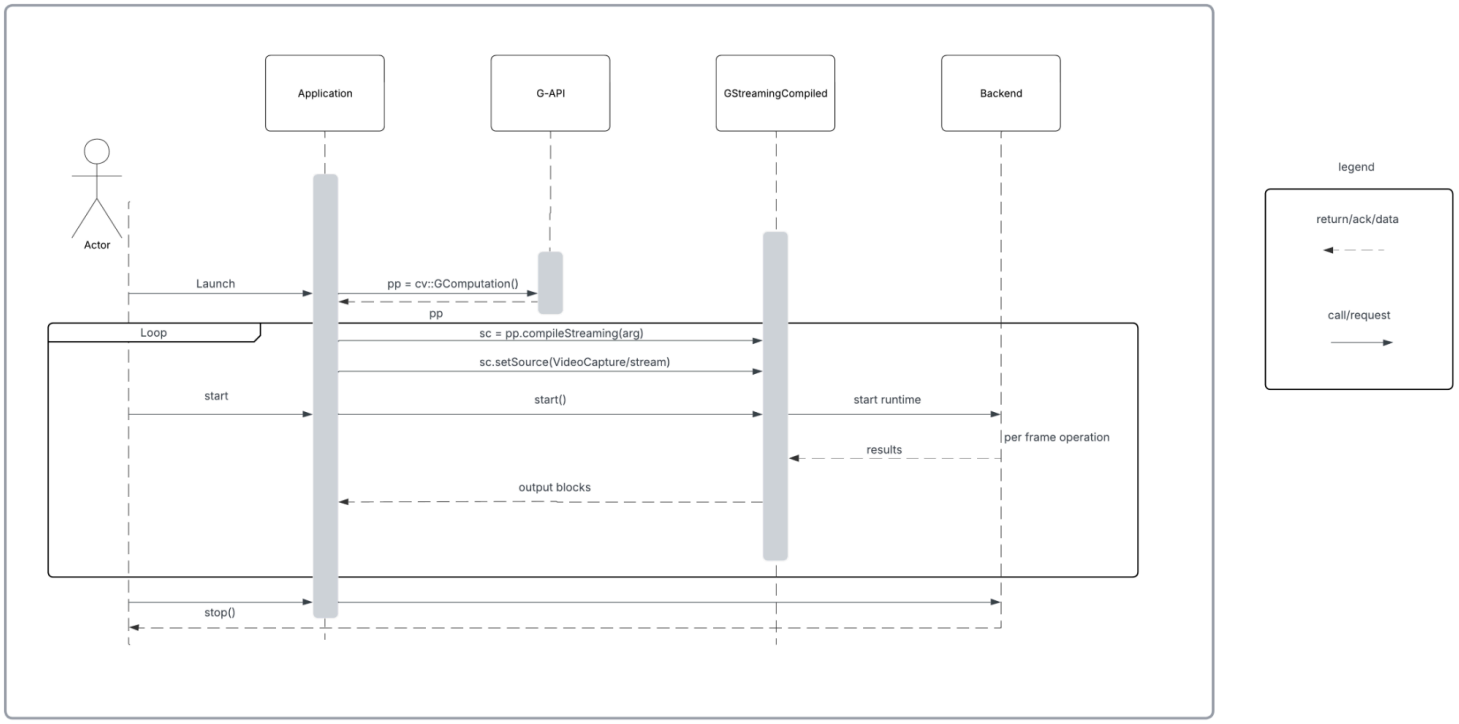


Figure 1: Streaming lifecycle for Webcam Object Detection with Overlay

Here is a sequence diagram of a use case of G-API where the application declares a graph that processes frames. The application declares the graph via cv::GComputation, compiles it for streaming with compileStreaming(args) (selecting a CPU/GPU/accelerator Backend), binds the camera source(example, could be any video input streaming source), and starts execution [9]. Results are returned on demand by a blocking pull(outputs) loop. Termination calls stop() for teardown. The declaration of the graph is first where the application constructs a GComputation and pp (a plan object) is returned [9]. No threads are started yet. The next step is the application supplies the live input via setSource method. The start() is invoked by the application lifeline. Each call returns the output blocks which are returned to the Application. The Application calls stop() when teardown is necessary or requested by the user, releases the source. A completion *ACK* of stop() is returned.

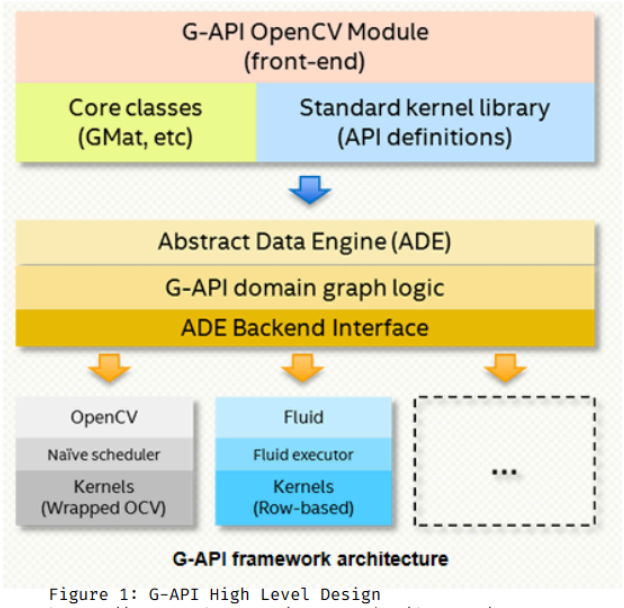


Figure 1: G-API High Level Design
https://www.ccoderun.ca/programming/doxygen/opencv/gapi_hld.html

Figure 2: Original Conceptual Diagram of G-API from the internet

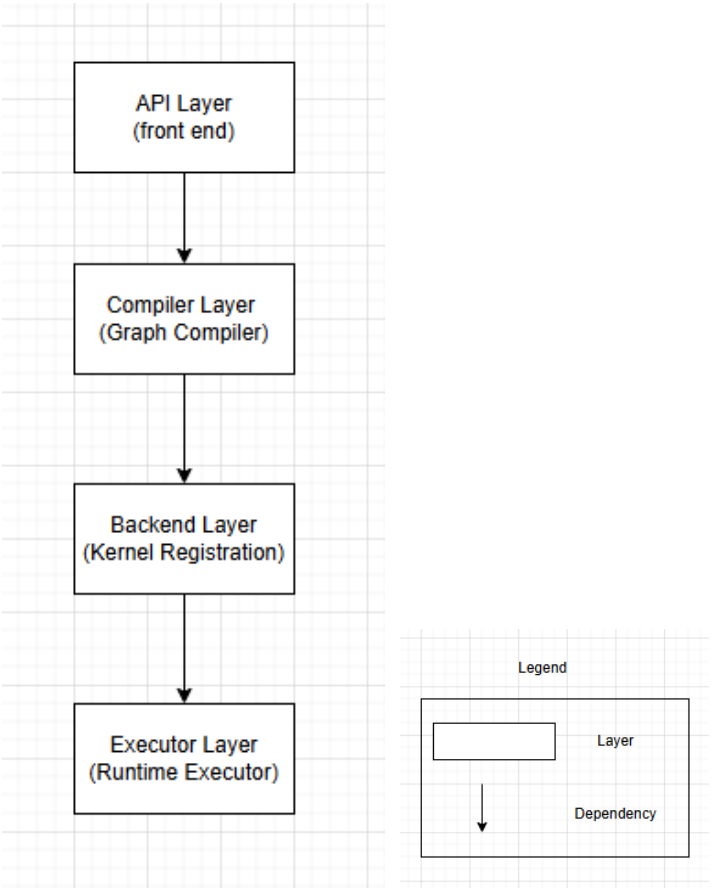


Figure 3: Reworked Conceptual Diagram using Extracted Dendencies

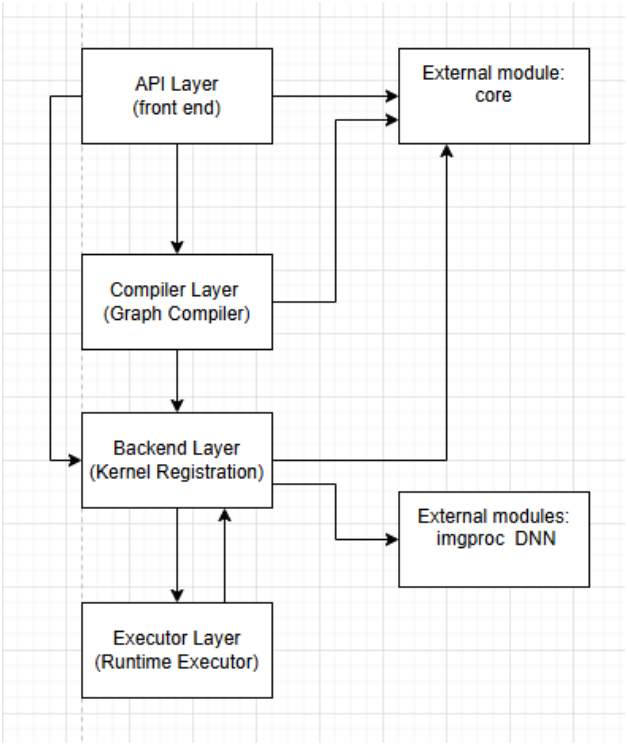


Figure 4: Mapping of Concrete Dependencies

The following design patterns of G-API work together as a cohesive system. The Facade pattern provides a simple interface for users to define and execute tasks without exposing subsystem complexity. Its responsibilities include accepting user input (graph definition, parameters) and then delegates detailed work to internal subsystems (Builder, Strategy, etc.). The Builder constructs the Processing Graph by incrementally building a compiled computation pipeline from parts. Once the graph is built, the Builder consults the Strategy layer to decide how this graph should be executed. The Strategy Pattern selects the most appropriate algorithm or backend implementation. It analyzes the available hardware (CPU, GPU, etc.), picks the right backend (e.g., OpenCL vs CPU), and defines execution policies. Once a Strategy is chosen, it directs the Factory to create the right type of objects that implement that strategy. The Factory pattern instantiates concrete components according to the chosen Strategy, creates backend-specific kernel or operation instances, and ensures that products share a common interface like the G-Kernel. Each created object becomes part of a command to be executed later. The Command pattern encapsulates executable actions by representing each operation as an independent, executable command object. Each graph node (like “blur” or “resize”) is a command that wraps the kernel created by the Factory. The runtime iterates through these Commands and invokes them using the selected Strategy.

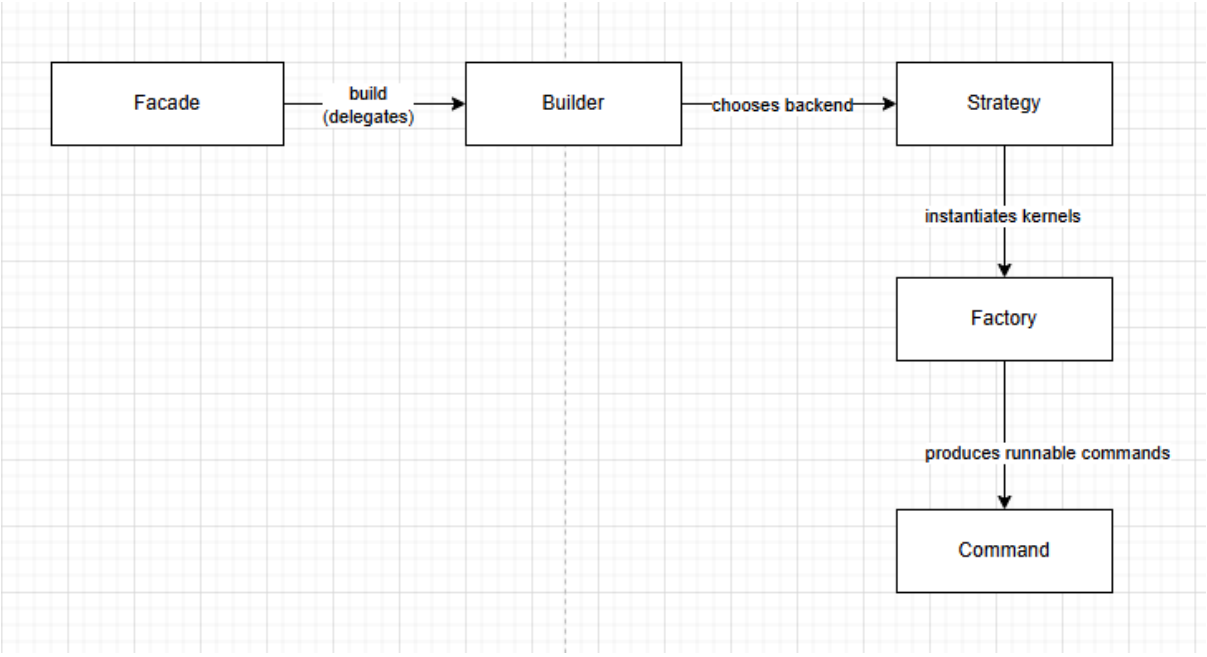


Figure 5: Interactions of design patterns

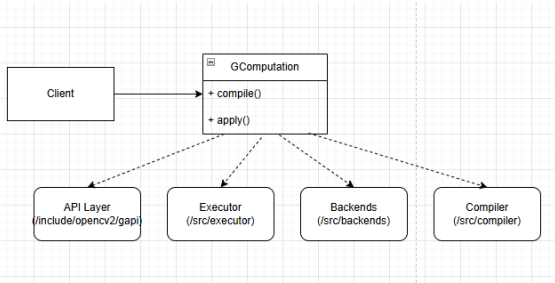


Figure 6: Facade Pattern

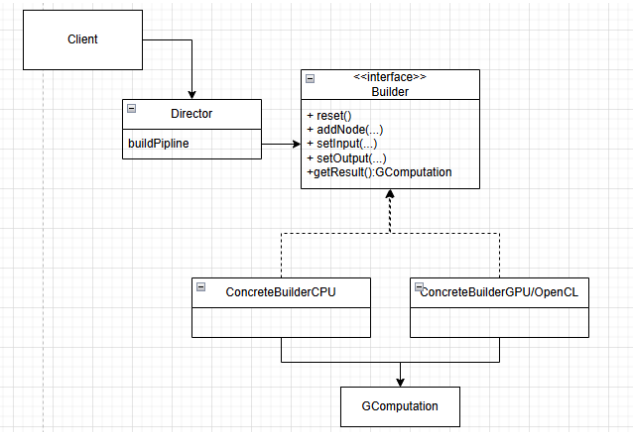


Figure 7: Builder Pattern

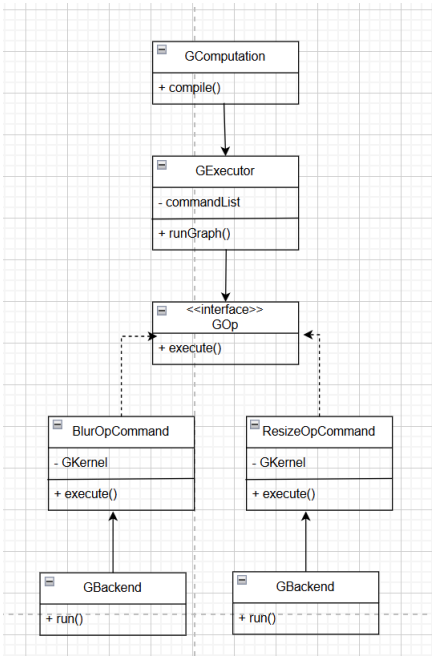


Figure 8: Command Pattern

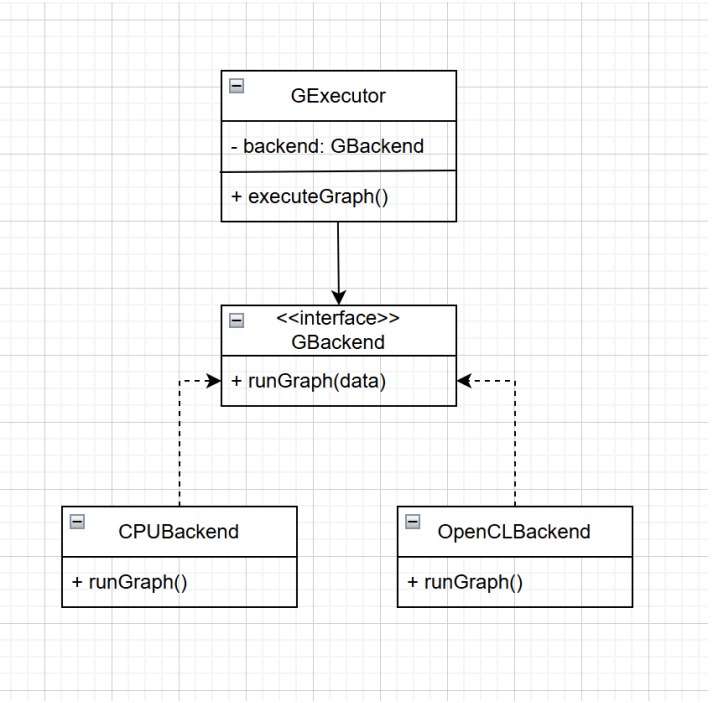


Figure 9: Strategy Pattern

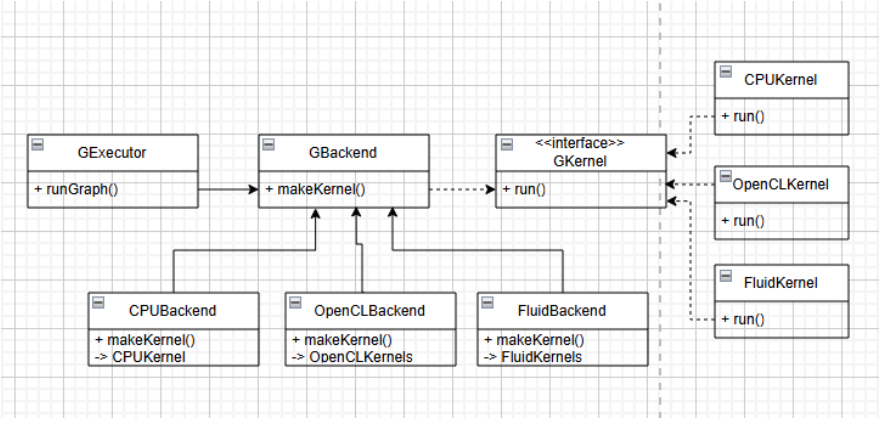


Figure 10: Factory Pattern

Conclusion

In conclusion, The overall concrete architecture we derived supported our conceptual architecture of openCV being modular and layered; and the top level subsystem G-API represents a modern, modular, and extensible design within OpenCV that leverages graph-based computation to achieve high performance and portability across hardware.

Lessons Learned

The G-API subsystem is divided into abstractions: the API, Compilation, Backend, and Execution. This allows for maintainability, extensibility, and easier debugging.

We learned that representing image processing operations as computation graphs allows deferred execution, global optimizations, and backend agnostic design.

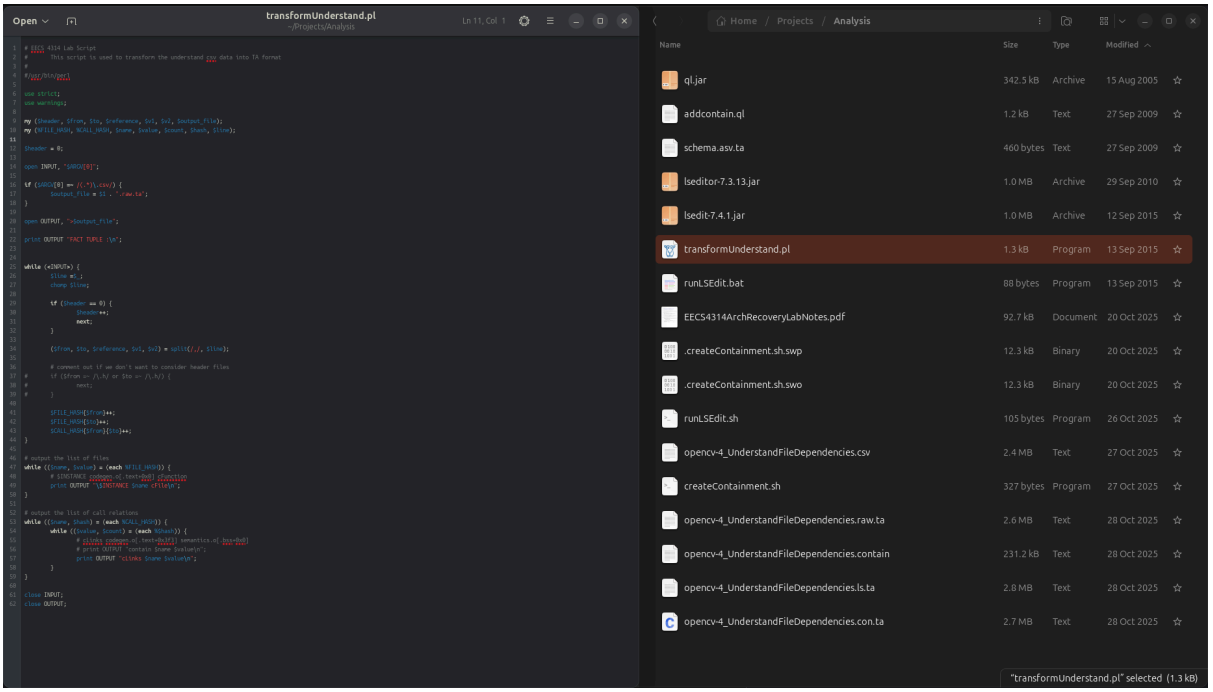
We also learned that the design patterns enable scalability, as the patterns together support flexible and high-performance architecture. Like the backend abstraction; that each backend registers its own kernels via factories and that new backends can be added without changing the front end API. Finally, we learned how to use understand and LSedit.

References

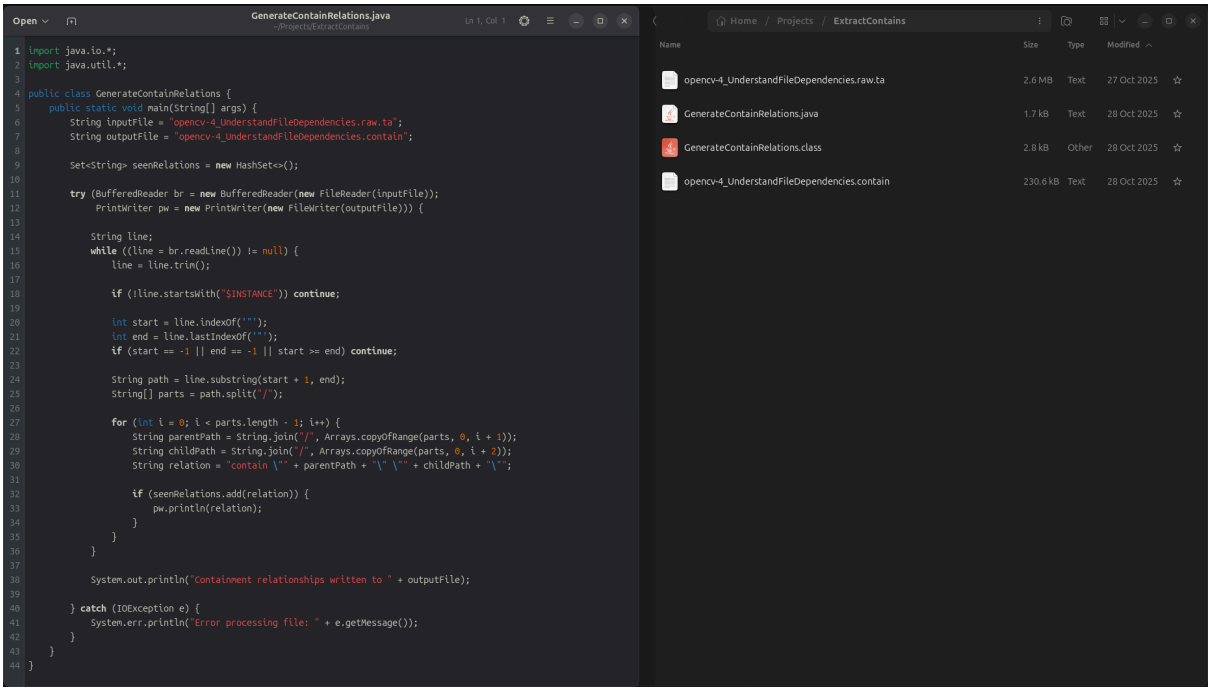
Architecure Derivation Process:

[2] JetBrains: *Compilation Database*. www.jetbrains.com/help/clion/compilation-database.html.

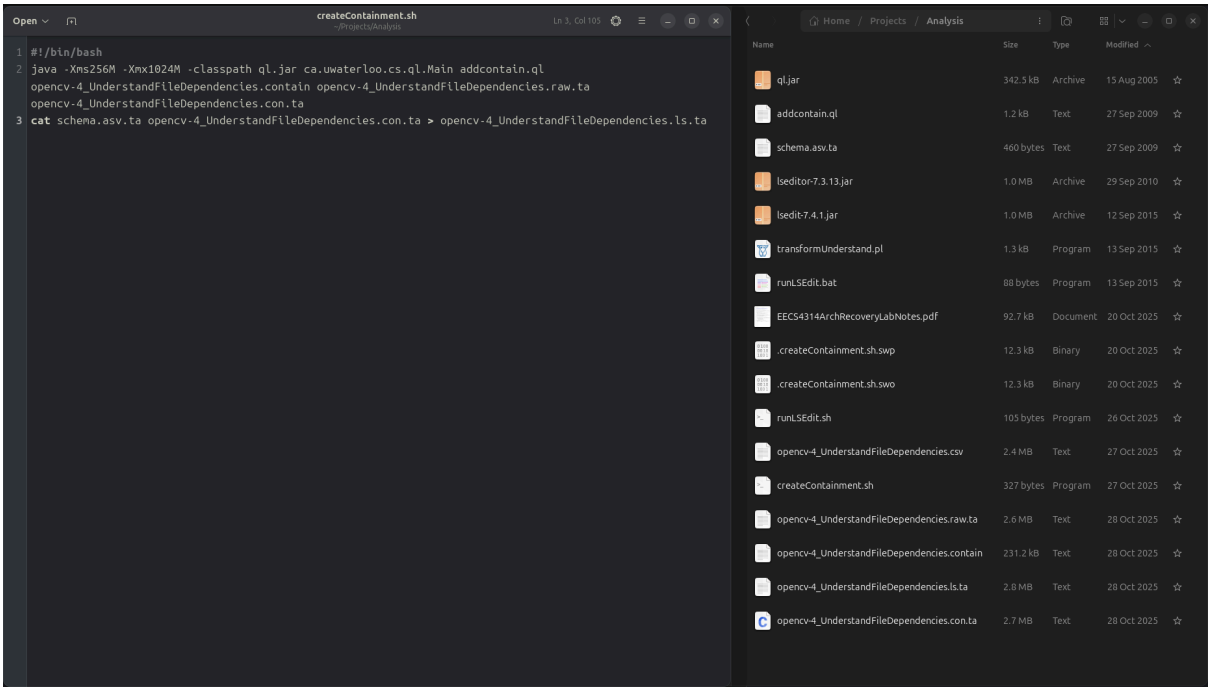
[3] eLand's modified perl script "transformUnderstand.pl" to create our TA file.



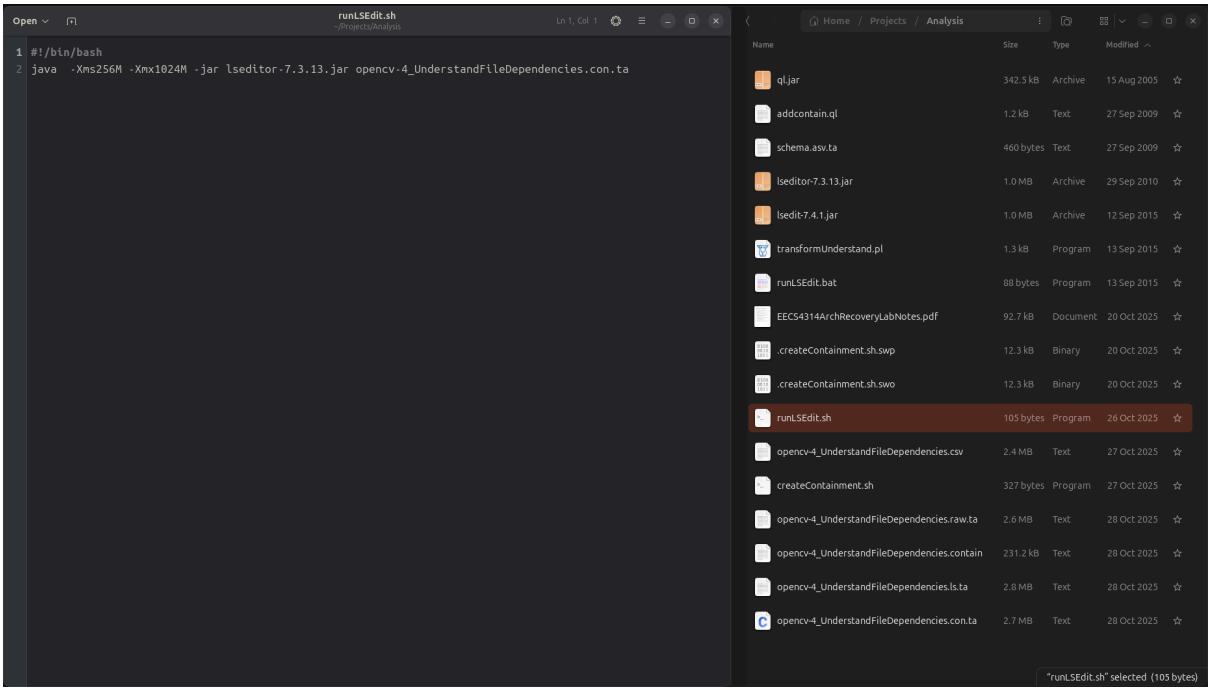
[4] eLand's Java script that created our custom containment file.



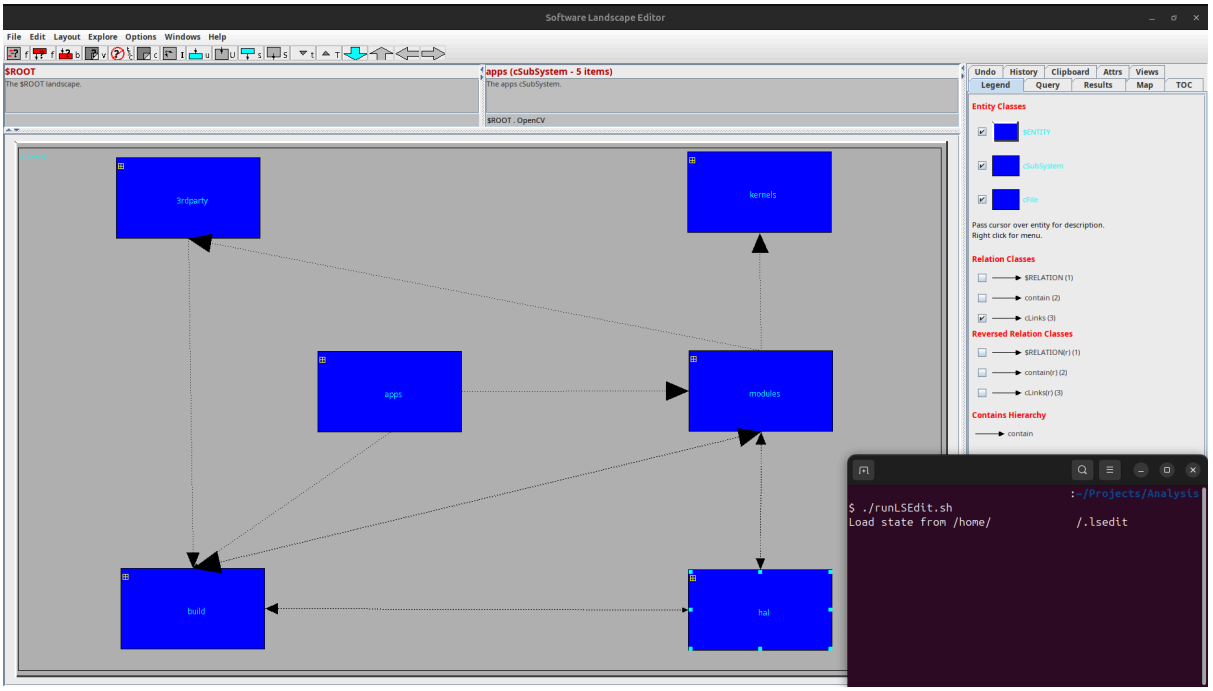
[5] eLand's modified script "createContainment.sh", used to impose our containment onto our extracted data.



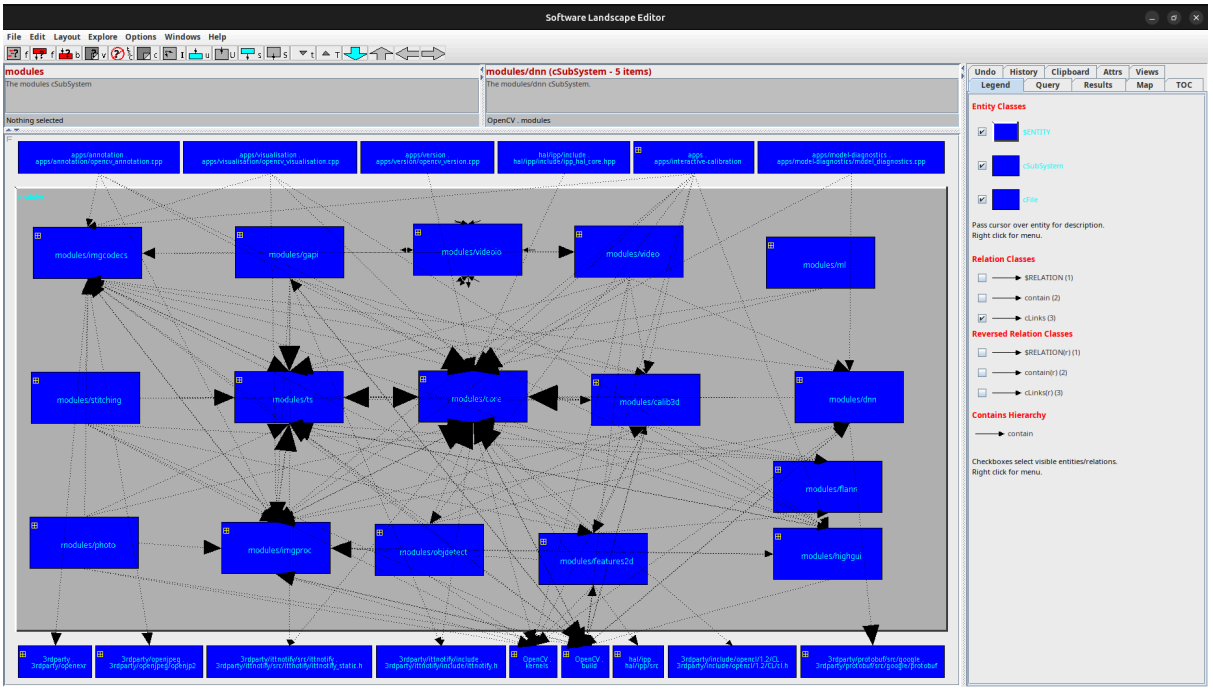
[6] eLand's modified shell script "runLSEdit.sh", used to visualize the extracted system architecture of OpenCV in LSEdit.



[7] OpenCV's top-layer subsystems and their inter-dependencies.



[8] OpenCV's main modules, visualised in LSEdit.



External interfaces:

[9] OpenCV Documentation: *G-API High-level design overview*. https://docs.opencv.org/4.x/de/d4d/gapi_hld.html

[10] OpenCV Documentation: *GRAPH-API*. https://docs.opencv.org/4.x/d0/d1e/gapi.html

[11] CCoderun Doxygen Reference: *OpenCV G-API*. https://www.ccoderun.ca/programming/doxygen/opencv/gapi.html

[12] OpenCV Documentation: *G-API Graph Compilation Arguments*. https://docs.opencv.org/4.x/d9/d29/group__gapi__compile__args.html

[13] Conan Blog: *OpenCV 4.0.0 new Graph API (G-API)*. https://blog.conan.io/2018/12/19/New-OpenCV-release-4-0.html

[14] OpenCV Documentation: *Graph API Tutorials*. https://docs.opencv.org/4.x/df/d7e/tutorial_table_of_content_gapi.html

[15] OpenCV Documentation: *Face analytics pipeline with G-API*. https://docs.opencv.org/4.x/d8/d24/tutorial_gapi_interactive_face_detection.html