Google Summer of Code 2026 - OSGeo Application

JSON Handling Toolkit for the QGIS Processing Framework

Sionigdha Sadhukhan | snigdha.lee75@gmail.com | github.com/Sionigdha

1. Contact Details

Name and Surname: Sionigdha Sadhukhan

Nickname: Sionigdha

Country: India

Email: snigdha.lee75@gmail.com

Phone: +91-8420871910

Public repositories:

https://github.com/Sionigdha

https://github.com/OSGeo/gdal/pulls?q=author%3ASionigdha+is%3Amerged

https://github.com/qgis/QGIS/pulls?q=author%3ASionigdha+is%3Amerged

LinkedIn: www.linkedin.com/in/sionigdha-sadhukhan-564b48218

Timezone: IST (UTC+5:30)

2. The Idea

OSGeo Member Software and Project Title

Organisation: QGIS / OSGeo

Project Title: JSON Handling Toolkit for the QGIS Processing Framework

Brief Description

The QGIS processing framework lacks a dedicated and comprehensive set of native algorithms for working with JSON data. There is no built-in way to load a JSON file into a model, extract a value from a JSON string, convert a JSON array to a vector layer, or serialise a vector layer's attributes back to JSON. This project fixes that with six new C++ processing algorithms: **LoadJSONFile, ExtractJSONValue, FlattenJSON, JSONToTable, JSONFromAttributes**, and **MergeJSONArrays.** Together they make JSON a first-class workflow component within the processing framework , any model, any algorithm output, any external data source that produces JSON can connect into the toolkit directly, with no custom scripting. Each algorithm is implemented as a QgsProcessingAlgorithm subclass in the QGIS native algorithm provider using nlohmann::json via QgsJsonUtils, which is already bundled in QGIS core and widely used across the codebase for JSON handling. All six will be written in C++. If required, I will draft and submit a QEP during the community bonding period before coding begins.

State of the Software Before GSoC

There is no dedicated set of processing algorithms for JSON data in the QGIS processing framework. Whenever JSON enters a QGIS workflow from a REST API response, from WFS metadata, from a plugin that writes structured output the only way to do anything with it inside a model is to write a custom Script algorithm in Python. That script is not reusable by others, it does not appear as a proper named node in the model designer, and it carries no documentation that another user can find. Multiply this across every team using QGIS for data pipelines and the same boilerplate JSON-parsing script gets written over and over.

3 concrete gaps illustrate the problem. If a model pulls a REST API response as a JSON string and needs to extract a single field value to drive a conditional branch, there is no processing algorithm for that you write json.loads() and a dictionary lookup in a custom script. If a workflow receives two JSON arrays from separate processing steps and needs to combine them before passing the result downstream, there is no processing algorithm for that either. And if you want to convert a vector layer's attribute table to JSON to feed an external API, there is no processing algorithm for that either. The framework simply has no vocabulary for JSON data at any point in the pipeline.

What This Project Will Bring

After this project, JSON is a first-class workflow component within the QGIS processing framework. A model can load a JSON file from disk, extract any value from it by key path, flatten an unknown structure to inspect its contents, convert an array to a vector layer, serialise a vector layer back to JSON, and merge two JSON arrays all using native algorithms with no custom scripting. The six algorithms together cover the complete JSON data lifecycle inside a model.

As one concrete illustration: a model that previously needed a custom Python script to extract a field value from a JSON string and drive a conditional branch can now do the same with LoadJSONFile into ExtractJSONValue into a Conditional Branch all native algorithms, no scripting. This works for any JSON source: a file on disk, a REST API response, a WFS metadata call, a plugin output. The toolkit is not anchored to any particular source of JSON. It works wherever JSON appears in a workflow.

LoadJSONFile : reads a JSON file from disk using QgsProcessingParameterFile input, parses it with nlohmann::json::parse() wrapped in a try/catch on json::parse_error, and returns QgsProcessingOutputString. Error handling covers file-not-found, encoding failures, malformed JSON, and empty files, each producing a QgsProcessingException with a specific message.

ExtractJSONValue : takes a JSON string and a dot-path key, parses the string with nlohmann::json::parse(), then traverses the resulting nlohmann::json object using key access for object nodes and integer index access for array nodes. Runtime type detection using .is_number_integer(), .is_number_float(), .is_string() determines whether to return QgsProcessingOutputString or QgsProcessingOutputNumber. Missing key paths raise QgsProcessingException naming the path that failed.

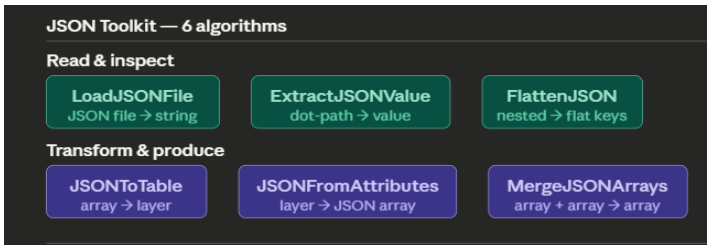
FlattenJSON : recursively flattens a nested JSON document by parsing with nlohmann::json::parse() and traversing the result using .items() for object nodes and integer indexing for array nodes, building a flat nlohmann::json object of dot-separated key paths to leaf values, and returning QgsProcessingOutputString. The output key paths map directly to

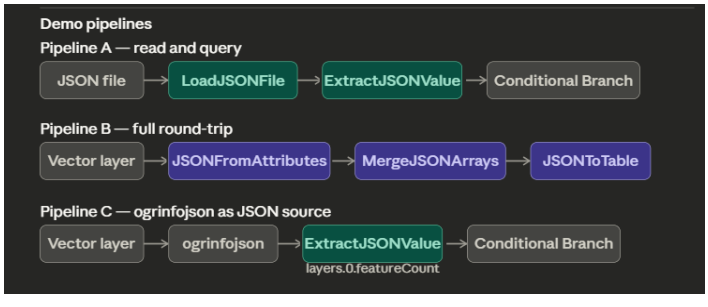
what ExtractJSONValue expects, making the two algorithms natural companions for inspecting unknown JSON structures.

JSONToTable : converts a JSON array to a vector layer. It parses the input with nlohmann::json::parse(), validates the top-level value is an array using .is_array(), performs a full-array scan for the union of keys, and uses .is_number_integer(), .is_number_float(), .is_string(), .is_boolean(), and .is_null() for type detection to construct QgsFields with QMetaType::Type::Int for integers, QMetaType::Type::Double for floats, QMetaType::Type::QString for strings, and QMetaType::Type::Int as 0/1 for booleans due to QGIS field type constraints. Missing fields are populated as NULL. Non-array top-level input raises QgsProcessingException. Returns QgsProcessingOutputVectorLayer.

JSONFromAttributes: the inverse of JSONToTable. Takes a QgsProcessingParameterVectorLayer input and an optional QgsProcessingParameterField for field selection, constructed with parentLayerParameterName pointing to the vector layer parameter so the field list populates correctly in the model designer. processAlgorithm() opens a QgsFeatureIterator and calls QgsJsonUtils::exportAttributesToJsonObject() per feature, which returns a JSON-compatible representation which can be directly used with nlohmann::json. All per-feature objects are pushed into a nlohmann::json array via push_back(), serialised to string via .dump(), and returned as QgsProcessingOutputString. This completes the round-trip: a vector layer can now move in and out of JSON format entirely within native QGIS algorithms.

MergeJSONArrays : takes 2 QgsProcessingParameterString inputs each containing a JSON array, parses both with nlohmann::json::parse(), validates that both top-level values are arrays using .is_array() (raising QgsProcessingException with a clear message if either is not), concatenates by iterating the second array and appending each element to the first via push_back(), serialises via .dump(), and returns QgsProcessingOutputString. Directly useful when combining JSON arrays from multiple processing steps or multiple API responses within a single model.





All six algorithms will be registered in qgsnativealgorithms.cpp via addAlgorithm() and their .cpp files added to QGIS_ANALYSIS_SRCS in src/analysis/CMakeLists.txt, following the exact pattern used by all existing native algorithms.

Future Developments

The most natural next step is a dedicated QgsProcessingParameterJSON type in C++ core. Right now JSON strings pass as QgsProcessingOutputString, which works but gives the model designer no way to know the output is specifically JSON. A proper JSON parameter type would enable validation, autocomplete for key paths, and better error messages. That is a C++ core change out of scope for this project, but the six-algorithm toolkit built here makes a strong practical case for it. JSONToTable could be extended to support NDJSON. FlattenJSON could gain a depth limit parameter. ExtractJSONValue could support JSONPath expressions. JSONFromAttributes could gain a geometry serialisation option to produce GeoJSON-compatible output directly.

3. Timeline

The project delivers 6 C++ processing algorithms across 14 coding weeks, structured in 3 phases.

Phase 1 (Weeks 1–3, May 27 – June 16) implements the read and inspect half of the toolkit: LoadJSONFile in Week 1 (May 27–June 2), ExtractJSONValue in Week 2 (June 3–9), and FlattenJSON in Week 3 (June 10–16), closing with the first PR.

Phase 2 (Weeks 4–6, June 17 – July 7) implements the transform and produce half: JSONToTable in Week 4 (June 17–23), JSONFromAttributes in Week 5 (June 24–30), and MergeJSONArrays in Week 6 (July 1–7), closing with the second PR.

Phase 3 (Weeks 7–14, July 8 – September 1) covers quality and delivery: Week 7 (July 8–14) is dedicated stress testing of all 6 algorithms, Week 8 (July 15–21) is integration testing and midterm evaluation, Week 9 (July 22–28) builds and validates 3 demo models, Week 10 (July 29–August 4) is a buffer for PR review cycles, Weeks 11–12 (August 5–18) cover integration and unit testing, Week 13 (August 19–25) is documentation, and Week 14 (August 26–September 1) is final review and submission.

Academic Conflicts

3 academic periods affect availability and are planned for explicitly. May 1 to May 9 is the end-semester theory examination period (100 marks, the heaviest period of the year), availability is reduced from April 13 to May 9. July 6 is the start of the odd semester, after which university runs alongside coding for the rest of the project. August 10 to August 18 is Term I theory examinations (30 marks, lighter) , testing and documentation work is intentionally placed in this window because it can be done in shorter focused sessions, unlike implementation which needs long uninterrupted blocks. Outside these windows, I treat this project as a full-time commitment and have no other jobs, internships, or planned vacations during the coding period.

Community Bonding Period (May 1 to May 26)

May 1–9 (Exams): QGIS is already built locally and the development environment is fully configured. Focus is on reading qgsalgorithmbuffer.h and qgsalgorithmbuffer.cpp in detail to understand the full C++ algorithm pattern how initAlgorithm() declares parameters, how processAlgorithm() reads them back via parameterAsString(), parameterAsInt(), and parameterAsDouble(), how createInstance() works, and how registration in qgsnativealgorithms.cpp and QGIS_ANALYSIS_SRCS in src/analysis/CMakeLists.txt is structured.

May 10–26 (Full availability): Focus shifts to 4 parallel tracks. First, studying nlohmann::json ,parse(), .items(), .is_array(), .is_object(), type detection methods, and .dump() , validated through small standalone C++ programs covering dot-path traversal, type inference, and array concatenation before touching the QGIS codebase. Second, studying QgsJsonUtils::exportAttributesToJsonObject() in detail to prepare for JSONFromAttributes. Third, drafting the QEP and submitting it for community review if required. Fourth, agreeing final scope, provider placement, and milestone targets with Valentin before Week 1 begins.

Week 1 (May 27 to June 2) LoadJSONFile

qgsalgorithmloadjsonfile.h and qgsalgorithmloadjsonfile.cpp are created. initAlgorithm() declares a QgsProcessingParameterFile input using Qgis::ProcessingFileParameterBehavior::File and a QgsProcessingOutputString output. processAlgorithm() reads the file and calls nlohmann::json::parse() inside a try/catch on json::parse_error file not found, encoding errors, malformed JSON, and empty file each produce a QgsProcessingException with a specific message. The .cpp file is added to QGIS_ANALYSIS_SRCS in src/analysis/CMakeLists.txt and the algorithm is registered in qgsnativealgorithms.cpp. The week closes by verifying the algorithm appears in the model designer and its output is connectable to downstream algorithms.

Week 2 (June 3 to June 9) ExtractJSONValue

qgsalgorithmextractjsonvalue.h and qgsalgorithmextractjsonvalue.cpp are created. The dot-path parser is implemented using nlohmann::json: after parsing the input string with nlohmann::json::parse(), each dot-separated token is used as a key for object nodes and converted to an integer index for array nodes. Runtime type detection via .is_number_integer(), .is_number_float(), and .is_string() determines whether to return QgsProcessingOutputString or QgsProcessingOutputNumber. Missing key paths raise QgsProcessingException naming the failed path. The algorithm is registered in the provider and CMakeLists. An end-to-end test connects LoadJSONFile into ExtractJSONValue in a live model using a real JSON fixture to confirm the full read-and-query chain works.

Week 3 (June 10 to June 16) FlattenJSON and First PR

qgsalgorithmflattenjson.h and qgsalgorithmflattenjson.cpp are created. The recursive flattening function parses the input with nlohmann::json::parse() then traverses the result using .items() for object nodes and integer indexing for array nodes, building a flat nlohmann::json object of dot-separated key paths to leaf string values, serialised via .dump() and returned as QgsProcessingOutputString. The algorithm is registered in the provider and CMakeLists. Testing against real JSON fixtures confirms that key paths produced by FlattenJSON match exactly what ExtractJSONValue expects, validating the 2 algorithms as a natural pair. A PR covering LoadJSONFile, ExtractJSONValue, and FlattenJSON is opened at the end of the week.

Week 4 (June 17 to June 23) JSONToTable

qgsalgorithmjsontotable.h and qgsalgorithmjsontotable.cpp are created. The implementation parses the input with nlohmann::json::parse(), validates it is an array using .is_array(), performs a full-array scan for the union of keys, and uses .is_number_integer(), .is_number_float(), .is_string(), .is_boolean(), and .is_null() for per-value type detection to construct QgsFields with QMetaType::Type::Int for integers, QMetaType::Type::Double for floats, QMetaType::Type::QString for strings, and QMetaType::Type::Int as 0/1 for booleans due to QGIS field type constraints. QgsFeature instances are populated with NULL for any key absent from a given record. Non-array top-level input raises QgsProcessingException. The algorithm is registered in the provider and CMakeLists. The output layer is verified to appear in QGIS with a correct attribute table.

Week 5 (June 24 to June 30) JSONFromAttributes

qgsalgorithmjsonfromattributes.h and qgsalgorithmjsonfromattributes.cpp are created. initAlgorithm() declares a QgsProcessingParameterVectorLayer input and an optional QgsProcessingParameterField for field selection, constructed with parentLayerParameterName pointing to the vector layer parameter so the field list populates correctly in the model designer. processAlgorithm() opens a QgsFeatureIterator and calls QgsJsonUtils::exportAttributesToJsonObject() per feature, which returns a JSON-compatible representation which can be directly used with nlohmann::json. All per-feature objects are pushed into a nlohmann::json array via push_back(), serialised via .dump(), and returned as QgsProcessingOutputString. The algorithm is registered in the provider and CMakeLists. A round-trip test confirms that a vector layer converted by JSONFromAttributes and then passed to JSONToTable produces an attribute table matching the original.

Week 6 (July 1 to July 7) MergeJSONArrays and Second PR

qgsalgorithmmergejsonarrays.h and qgsalgorithmmergejsonarrays.cpp are created. initAlgorithm() declares 2 QgsProcessingParameterString inputs. processAlgorithm() parses both with nlohmann::json::parse(), validates that both top-level values are arrays using .is_array(), concatenates by iterating the second array and appending each element to the first via push_back(), serialises via .dump(), and returns QgsProcessingOutputString. Non-array input on either parameter raises QgsProcessingException with a clear message. The algorithm is registered in the provider and CMakeLists. A PR covering JSONToTable, JSONFromAttributes, and MergeJSONArrays is opened at the end of the week.

Week 7 (July 8 to July 14) Stress Testing All Six Algorithms

All 6 algorithms are stress-tested against difficult inputs. JSONToTable: arrays of 1000 or more items, heterogeneous schemas, empty arrays, all-null columns. JSONFromAttributes: layers with many fields, layers with null geometry, field subset selection. MergeJSONArrays: 1 empty array and 1 non-empty, both empty, arrays containing nested objects. ExtractJSONValue: deeply nested paths, paths pointing to objects rather than leaf values, paths with numeric indices into nested arrays. FlattenJSON: deeply nested structures, arrays of arrays, objects with empty string keys. LoadJSONFile: files with BOM, Windows line endings, very large files. Edge cases found during this week are fixed. Review comments on both open PRs are addressed.

Week 8 (July 15 to July 21) Integration Testing and Midterm

All 6 algorithms are tested together in end-to-end pipelines using real data: JSON files from disk, REST API response fixtures, GeoJSON feature collections, and synthetic vector layers. The full QGIS algorithm test suite is run locally to confirm no regressions introduced by the new registrations in qgsnativealgorithms.cpp. The midterm evaluation check-in with Valentin is completed this week, presenting working demonstrations of all 6 algorithms in the model designer.

Week 9 (July 22 to July 28) Demo Models

3 demo models are built and validated in the QGIS model designer. Pipeline A demonstrates the core read-and-query workflow: a JSON file on disk goes through LoadJSONFile, ExtractJSONValue pulls a value by dot-path key, and the result feeds a Conditional Branch. Pipeline B demonstrates the full attribute round-trip: a vector layer goes through JSONFromAttributes, the output is merged with a second JSON array via MergeJSONArrays, and the merged array is converted back to a vector layer by JSONToTable. Pipeline C demonstrates the toolkit working with an existing QGIS algorithm as a JSON source: a vector layer goes through ogrinfojson, ExtractJSONValue pulls layers.0.featureCount from the output, and the result drives a Conditional Branch showing that the toolkit connects naturally to any algorithm that produces JSON, not just external sources. All 3 are saved as .model3 files for inclusion in the repository and tested against real shapefiles.

Week 10 (July 29 to August 4) Buffer and PR Review

This week is reserved for addressing any outstanding review comments across both PRs and handling any implementation issues surfaced during demo model testing. If both PRs are already merged, the time goes to writing additional test fixtures and improving inline code documentation. No new features are added this week exists because C++ review cycles in QGIS can run longer than expected and the project needs a buffer before the testing phase begins.

Week 11 (August 5 to August 11) Integration Tests

Integration tests for all 6 algorithms are written in the QGIS C++ algorithm test suite, following the patterns used by existing native algorithm tests. Each algorithm is tested against the full range of valid and invalid inputs established during stress testing in Week 7. The full test suite is run locally to confirm all tests pass and no regressions exist. University Term I exams begin August 10 , these are light 30-mark tests, and integration test writing is placed here deliberately because it can be done in shorter focused sessions that fit around exam preparation.

Week 12 (August 12 to August 18) Unit Tests

C++ unit tests are written for all 6 algorithms using the QGIS test framework, QObject-based test classes compiled against the QGIS analysis library, following the same structure as existing tests in tests/src/analysis/. LoadJSONFile: valid UTF-8 JSON, malformed JSON triggering json::parse_error, file not found, empty file, file with BOM. ExtractJSONValue: nested object traversal, array index access, missing key raising QgsProcessingException, path to non-leaf node, numeric versus string type detection. FlattenJSON: nested objects, arrays of objects, empty object, mixed nested structures. JSONToTable: correct feature count, field derivation from union of keys, type inference for all supported QMetaType values, empty array input, mixed-schema arrays, non-array top-level input raising QgsProcessingException.

JSONFromAttributes: correct field count in output JSON, null attribute handled correctly by QgsJsonUtils::exportAttributesToJsonObject(), field subset selection, round-trip consistency with JSONToTable. MergeJSONArrays: correct element count after merge, non-array first input exception, non-array second input exception, both-empty-array case.

Week 13 (August 19 to August 25) Documentation

User-facing documentation is written for all six algorithms following the standard QGIS algorithm documentation format used in the existing algorithm reference. Each algorithm entry covers purpose, parameters with types and constraints, outputs, and at least one concrete example. A worked example narrative is written for all three demo models explaining the end-to-end flow from raw input to final output, aimed at a QGIS user who has not read the source code.

Week 14 (August 26 to September 1) Final

All remaining review comments across all PRs are addressed. Final polish ensures all tests pass on CI. The GSoC final evaluation is submitted.

4. Studies

Degree: B.Tech in Computer Science and Engineering (IoT Specialisation)

Institution: Institute of Engineering and Management (IEM), Salt Lake, Kolkata, India

Year: Third year (2023–2027)

CGPA: 9.2 / 10.0

Higher Secondary (Class XII): Delhi Public School Ruby Park, Kolkata | CBSE (2023) | **94%**

Secondary (Class X): Delhi Public School Megacity, Kolkata | ICSE (2021) | **95%**

Contribution to Studies

The IoT specialisation centres on data pipelines, sensor streams, and typed data flow exactly the conceptual territory the QGIS processing framework occupies. The coursework covers how typed data moves through a system, how APIs are designed for data transformation, and how pipeline components are tested in isolation. Working on the QGIS processing framework is a direct extension of that material, but at a scale and in a codebase the coursework cannot provide. Reading, navigating, and contributing to a large production C++ codebase with real code review is an experience that classroom projects cannot replicate. This project gives that experience in a concrete, measurable way tied to work that ships into a production geospatial tool used worldwide.

5. Programming and GIS Experience

Computing Experience

Python is the strongest language. C++ (intermediate) is being actively learned. I have read and navigated the QGIS C++ codebase, understand the QgsProcessingAlgorithm pattern from initAlgorithm() through processAlgorithm() and createInstance(), and I have been actively building C++ familiarity through reading the QGIS codebase, studying the QgsProcessingAlgorithm pattern end to end, and writing standalone programs against nlohmann::json to validate the traversal and type detection logic before touching the QGIS codebase. This project will be my first merged C++ contribution to QGIS. Full-stack experience includes Node.js, Express, React, Next.js, and Flask. Comfortable with Git PR workflows, CMake, and conda environments. During an internship at iQuester, I worked with InfluxDB for time-series data ingestion and REST APIs for IoT sensor pipelines.

Prototype

To validate the implementation approach, I compiled a working prototype of LoadJSONFile locally against libqgis_analysis. It compiles cleanly following the exact QgsProcessingAlgorithm pattern used by existing native algorithms.

`qgsalgorithmloadjsonfile.h`:

```
#ifndef QGSALGORITHMLOADJSONFILE_H
#define QGSALGORITHMLOADJSONFILE_H

#include "qgis_sip.h"
#include "qgsprocessingalgorithm.h"

class QgsLoadJSONFileAlgorithm : public QgsProcessingAlgorithm
{
  public:
    QgsLoadJSONFileAlgorithm() = default;
    QString name() const override;
    QString displayName() const override;
    QStringList tags() const override;
    QString group() const override;
    QString groupId() const override;
    QString shortHelpString() const override;
    QgsLoadJSONFileAlgorithm *createInstance() const override SIP_FACTORY;

  protected:
    void initAlgorithm( const QVariantMap &configuration = QVariantMap() ) override;
    QVariantMap processAlgorithm( const QVariantMap &parameters,
                                  QgsProcessingContext &context,
                                  QgsProcessingFeedback *feedback ) override;
};

#endif
```

`qgsalgorithmloadjsonfile.cpp`:

```
#include "qgsalgorithmloadjsonfile.h"
#include "qgsprocessingparameters.h"
#include "qgsprocessingoutputs.h"
#include "qgsjsonutils.h"
#include <QFile>
#include <QTextStream>
#include <nlohmann/json.hpp>

using namespace nlohmann;

QString QgsLoadJSONFileAlgorithm::name() const
{ return QStringLiteral( "loadjsonfile" ); }

QString QgsLoadJSONFileAlgorithm::displayName() const
{ return QObject::tr( "Load JSON file" ); }

QStringList QgsLoadJSONFileAlgorithm::tags() const
{ return QObject::tr( "json,load,file,parse" ).split( ',' ); }

QString QgsLoadJSONFileAlgorithm::group() const
{ return QObject::tr( "JSON" ); }

QString QgsLoadJSONFileAlgorithm::groupId() const
{ return QStringLiteral( "json" ); }

QString QgsLoadJSONFileAlgorithm::shortHelpString() const
{ return QObject::tr( "Loads a JSON file from disk and outputs its contents as a string." ); }

QgsLoadJSONFileAlgorithm *QgsLoadJSONFileAlgorithm::createInstance() const
{ return new QgsLoadJSONFileAlgorithm(); }

void QgsLoadJSONFileAlgorithm::initAlgorithm( const QVariantMap & )
{
  addParameter( new QgsProcessingParameterFile(
    QStringLiteral( "INPUT" ),
    QObject::tr( "JSON file" ),
    Qgis::ProcessingFileParameterBehavior::File,
    QStringLiteral( "json" )
  ) );
  addOutput( new QgsProcessingOutputString(
    QStringLiteral( "OUTPUT" ),
    QObject::tr( "JSON string" )
  ) );
}

QVariantMap QgsLoadJSONFileAlgorithm::processAlgorithm(
  const QVariantMap &parameters,
  QgsProcessingContext &context,
  QgsProcessingFeedback *feedback )
{
  Q_UNUSED( feedback )
  const QString path = parameterAsFile( parameters, QStringLiteral( "INPUT" ), context );

  QFile file( path );
  if ( !file.open( QIODevice::ReadOnly | QIODevice::Text ) )
    throw QgsProcessingException( QObject::tr( "Could not open file: %1" ).arg( path ) );

  QTextStream stream( &file );
  const QString content = stream.readAll();

  try
  {
    json parsed = json::parse( content.toStdString() );
    QVariantMap result;
    result.insert( QStringLiteral( "OUTPUT" ), QString::fromStdString( parsed.dump() ) );
    return result;
  }
  catch ( json::parse_error &e )
  {
    throw QgsProcessingException(
      QObject::tr( "Invalid JSON: %1" ).arg( QString::fromStdString( e.what() ) )
    );
  }
}
```

The prototype uses `nlohmann::json::parse()` for parsing, `json::parse_error` for error handling, and `parsed.dump()` for serialisation the same three nlohmann calls , across all six proposed algorithms. `QgsJsonUtils` is included ready for JSONFromAttributes which uses `QgsJsonUtils::exportAttributesToJsonObject()`.

GIS Experience

Exposure to GIS came primarily through contributing to the QGIS and GDAL codebases rather than through formal coursework. I have explored the QGIS processing framework and Model Designer in depth to understand how algorithms connect, how vector layer inputs move through workflows, and how algorithm providers are structured. Before contributing to open source GIS projects, interaction with geospatial data was mainly through web projects that consumed GeoJSON.

GIS Programming and Open Source Contributions

I have been contributing to GDAL since early 2026. My contributions include implementing new CLI features in GDALVectorInfo, improving the test infrastructure by marking network-dependent tests correctly to prevent false CI failures, and adding documentation improvements such as reference tables and build fixes. One of my patches was also backported to the stable release branch.

The most substantial contribution was adding a new command-line flag to **gdal vector info**, which was developed with considerable help of GDAL core maintainer Even Rouault. During the review processes he provided his valuable guidance on community norms, testing practices, and how to structure changes for maintainability. This experience helped me understand how large open-source projects review contributions and maintain production-quality C++ code.

Full list of merged contributions:
https://github.com/OSGeo/gdal/pulls?q=author%3ASionigdha+is%3Amerged

On the QGIS side, I have contributed documentation improvements targeting the QGIS 4.0 development cycle. These contributions include clarifying the initialization behavior of the QgsLogger class and improving related documentation to make the behavior clearer for developers.
List of merged QGIS contributions:
https://github.com/qgis/QGIS/pulls?q=author%3ASionigdha+is%3Amerged

6. GSoC Participation

Have you participated in GSoC before? **No**, this is my first time applying.

Have you applied but were not selected? **No**.

Will you submit another proposal to a different org this year? **No**. This is my only GSoC application for 2026.