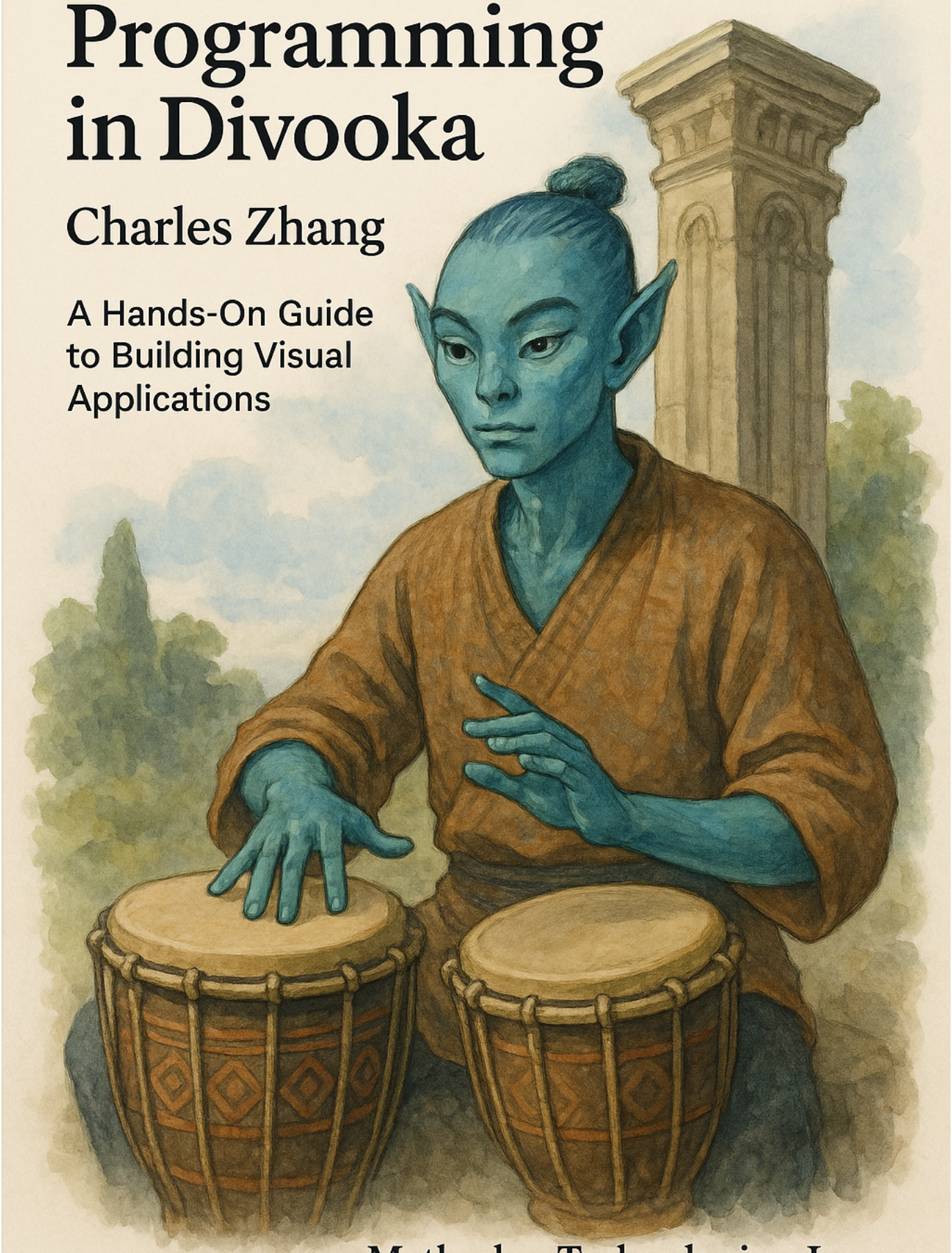
Introductory Visual Programming in Divooka

Charles Zhang

A Hands-On Guide to Building Visual Applications



Methodox Technologies, Inc.

Introductory Visual Programming in Divooka

Author: Charles Zhang
Published by: Methodox Technologies, Inc.

Edition: First Edition, 2025
Revision: 001

Preface

Welcome to Divooka! In this little book we are going to dive heads in to the practical usages of Divooka. After going through many topics in this book, you will be able to automate file system operations, analyze tabular data, chain LLM and AI operations together, and have a solid ground understanding of the Divooka ecosystem. From there, you can pick specific topics of interest and study deeper and sharpen your programming skills.

Divooka is a high-level visual programming language. There isn't as many general-purpose visual programming languages out there as Divooka, so most readers familiar with traditional C-style or similar text-based languages may not have heard of concept before. It comes down to dragging and connecting node graphs instead of writing codes for the most of the program construction - for the most, but not all. A good and highly efficient visual programming language is a hybrid between visual blocks and coding - taking the best of two worlds.

Divooka is designed with simplicity and robustness in mind. It's easy enough for beginners to pick up and get useful things done, and complex enough for a veteran developer to integrate into a larger existing software system.

This textbook is developed as a series of notes for my internal training and public training series during April to September 2025 and will serve as a foundation for introductory Divooka usage. The focus is on practical aspects rather than theory.

It's my wish that this little textbook will give the readers a fairly comprehensible and self-contained overview on the overall usage of Divooka. It's by no means intended to be comprehensive though - Divooka is a large system with many components and it will require a few books to cover it in sufficient detail.

How to Use This Book

This book assumes the Divooka Explore distribution as its primary context, though most of the knowledge presented here is also applicable to Divooka Engine and Divooka Compute. We have deliberately avoided topics that are specific to Divooka Compute or too low-level in nature to keep the content accessible and broadly useful.

Because Divooka is under active development and has not yet reached a final stable release, some of the content in this book is forward-looking and describes the "ideal" version of the platform. As such, certain APIs and features discussed may change over time. While we aim to keep this book updated regularly, for the most current and definitive information, please refer to the Methodox Wiki and the official Divooka API documentation.

Divooka is designed as a general-purpose platform that supports full-stack development, and a complete study of the system will touch on a wide range of disciplines. Depending on your role or interests - whether you are automating workflows, working in data analysis, developing backend systems, or designing user-facing applications - you may choose to focus on specific sections most relevant to your work.

The foundational chapters - particularly the introduction to the interface, usage conventions, and plugin system - are broadly applicable and worth reading in full. The chapters covering applied or domain-specific topics may be skimmed or selectively studied based on your professional focus.

For Data Analysts and Data Scientists, the book covers a number of high-level libraries and workflows geared toward practical data work. This includes tools for tabular data transformation and analysis, database querying, report generation, and automation. To support analytical applications, Divooka offers libraries on statistical modeling, probabilistic programming, and basic machine learning. Contents on those workflows are not included in this book - you may consult Methodox Wiki or The Definitive Guide textbook.

For System Administrators and Automation Engineers, the emphasis is on large-scale or repetitive task management. Key chapters include those on scripting and batch processing, command-line interfaces (CLI), and structured interaction with the file system. These tools are designed to assist with integration, monitoring, and control across a range of environments.

For AI and Creative Use Cases, Divooka provides a streamlined set of interfaces for working with modern AI tools and multimedia systems. Relevant chapters introduce APIs for large language models (LLMs), image and video processing, and integrations with online services, enabling rapid prototyping and content generation.

For Plugin Developers and Internal Tool Builders, the platform supports a modular development workflow. These users will benefit from the chapters on C# scripting, the Pure functional layer, and the full plugin lifecycle. Practical topics such as exposing new nodes, wrapping existing code for visual use, and CLI-based automation are also covered to support cross-team collaboration.

You can find additional examples, templates, and use-case guides on the Methodox Wiki (https://wiki.methodox.io/en/KnowledgeBase/Learning).

Training Instructions

For short-term training sessions spanning approximately 4–5 lessons, the recommended structure begins with foundational topics such as navigating the Divooka interface, understanding the graph-based workspace, and learning how to save and load documents. Once learners are comfortable with the environment, the sessions should pivot to a focused area of application - such as data analytics, image processing, or automation - based on the participants' backgrounds and interests. Introducing the concept of the procedural context early on will also help learners understand how data flows through the system and how different nodes interact.

For developer-focused cohorts, it is especially valuable to include content on CLI integration, scripting capabilities, and automation workflows. These features not only demonstrate Divooka's flexibility but also help connect it to existing infrastructure and tooling in professional environments.

Regardless of audience, hands-on experimentation should be emphasized. Encourage participants to actively explore and test features themselves during and between sessions. This interactive approach fosters stronger understanding and engagement compared to passive demonstration alone. This spirit of discovery and exploration is central to the Divooka experience - and often where learners begin to see its full potential in their own work.

For longer mini-courses (5–12 lessons), instructors may extend the curriculum to include additional modules such as plugin development, advanced visualization techniques, or API-based integrations. Supplementing the textbook with real-world mini-projects or role-specific case studies also enhances relevance and retention.

Table of Contents

Figures

(Figures and Pages)

Chapter 1. Getting Started

> In a world driven by data and automation, Divooka enables you to connect ideas as easily as building lego - visualize your workflow, define your logic, and let the platform handle the rest.

1.1 Distributions and Version Variations

Divooka programs are developed using "graph editors," which provide a visual environment for constructing workflows. Depending on the platform you use and preferred working style, there are three official flavors - **Neo**, **Gospel**, and **Morpheus** - each offering the same core functionality: a canvas where you drag, drop, and connect nodes to define program logic, ultimately saving your work into `.dvk` document files. Although these editors differ in visual theme and minor layout details, they share a consistent set of interface elements: a menu bar, a main canvas area, a node properties panel, and a tools tray. This uniformity ensures that once you learn one editor, you can switch to another with minimal friction.

(Neo main screen)

(Gospel main screen)

(Morpheus main screen)

Despite superficial differences in styling, the fundamental building blocks - or function blocks - in all three editors have nearly identical GUI components. For example, a typical node in Neo might look like the diagram below, with labeled input and output ports, a title bar, and a settings icon.

(Use Neo node as example for node surface GUI components breakdown)

Below is the same node rendered in Gospel; although the colors and fonts differ, the layout of inputs, outputs, and context menu remains consistent.

(Gospel node)

Why Different Editors?

Supporting multiple graph editors serves more than just offering different visual themes. At a basic level, each editor targets a different platform or user persona - Neo might be optimized for Windows desktop performance, Gospel for

cross-platform compatibility, and Morpheus for experimental features in development. More fundamentally, Divooka is built on a set of open specifications and standards that define how nodes, connections, and workflows are represented. By decoupling these standards from any particular editor implementation, Divooka guarantees that custom editors or third-party tools can interoperate with existing workflows. In practice, this means you could have a specialized editor for data analytics, another for IoT device control, and still share the same underlying `.dvk` files.

To define behavior within a workflow, you initiate a connection by clicking and dragging from an output connector on one node to an input connector on another. If you need to remove an existing connection, hold the **Alt** key while clicking either connector, or press **Alt** and use the scissor icon to cut the wire - provided that the editor you are using supports the scissor action.

(Show scissor cut in Morpheus editor)

1.2 Installation

Divooka can be used entirely online or installed locally on your machine. The online version is ideal for quickly experimenting with basic workflows or deploying cloud-based workflows, while the local installation is recommended when you need direct access to files on your computer or want to integrate with local services.

Try It Online

Visit tryitnow.divooka.com to launch the Divooka web interface in your browser. You will have unlimited free access for up to 30 minutes at a time; after that, simply refresh or log in with a **Methodox One** account to extend your session. The online environment includes most features but restricts where your data is stored (typically on cloud servers) and may be subject to rate limits if your workflows become very large.

(Screenshot of Try It Online interface)

Install with an Installer

For local usage, download the **Divooka Explore** installer (which is free) from our methodox.itch.io storefront. Follow these steps:

1. Visit the Divooka Explore page on Methodox Store and click "Download."
2. Choose the installer matching your operating system (Windows EXE, macOS PKG, or Linux Zip).
3. Run the downloaded installer and follow the on-screen prompts to accept the license agreement and pick an installation directory.
4. Once installation finishes, a desktop shortcut (or application icon in your launcher) will be created automatically.

(Download screenshot)

(Install screenshot)

(Install settings screenshot)

(Finished shortcut screenshot)

Double-click the **Divooka Explore** icon to launch the application and you will be greeted with the welcome screen, from which you can create or open existing workflows.

(Welcome screen)

Install from Zip

If you want access to the latest nightly builds or experimental platforms (e.g., ARM Linux), we distribute a plain `.zip` package instead of a full installer. To install from zip:

1. Unzip the downloaded file into any folder of your choice.
2. Locate the `DivookaExplore.exe` (Windows) or equivalent executable in the extracted directory.
3. Double-click to launch. You can create a desktop shortcut manually if desired.

(Screenshot of zip)

(Screenshot of unzipping)

(Screenshot of file folder and .exe in it)

Compile from Source

We plan to make Divooka's source code available under an open-source license in a future release. Once the repository is public, you can compile Divooka by running a PowerShell script (`build.ps1`) on Windows or using the `.NET CLI` (`dotnet publish`) on other platforms. The build output will include everything you need to run the graph editor locally.

Running from Docker

If you maintain Linux servers or prefer containerized deployments, we provide official Docker images for Divooka. For example:

- `divooka/stew:latest` - a headless CLI version for batch automation
- `divooka/gospel:latest` - GUI version accessible through a browser or VNC

Detailed Docker usage instructions are beyond the scope of this guide, but you can find them in the online documentation.

Run Divooka on Mobile Devices

We also offer Divooka Compute on mobile platforms. Download the Android APK from Google Play or the iOS app from the App Store. For simple "viewer" programs that let you open `.dvk` files without editing, download **Divooka Viewer** from the official site: `https://divooka.come/download`

1.3 Divooka Explore Distribution

Divooka ships in multiple **distributions**, each tailored to specific workloads. The **Divooka Explore** distribution is the community-focused edition, offering the most commonly used features under a permissive, non-commercial license. It bundles command-line utilities and helper applications such as **stew**, **Jarvis**, and **Divooka Viewer**.

Below is a high-level view of the **Divooka Explore** folder structure. Instead of listing every DLL, we group components by function to give you an overview of what comes included:

Folder structure at a glance[^1]

```
Divooka Explore/
├── Core Application     # Main EXE + DLL + dependency descriptors
├── Engine               # Shared engine component
├── Supporting Tools     # Additional executables (Glaze, Jarvis, stew, Dashboard)
├── Libraries            # All Divooka-specific and third-party DLLs
├── Examples             # Sample .dvk workflows (Daily Utility, Generative AI)
├── Licenses             # Third-party license files
├── Plugins              # Extendable plugin folder
└── UserManual           # End-user & developer documentation
```

- **Core Application & Engine**
  - `Divooka Explore.exe` launches the main graph editor UI.
  - `Divooka Engine.dll` contains the runtime logic that parses `.dvk` files, resolves node dependencies, and orchestrates execution.
- **Supporting Tools**
  - **Glaze**: A standalone tool for batch processing of Divooka workflows.
  - **Jarvis**: A personal desktop assistant application that can be customized using Divooka graphs.
  - **stew**: A CLI utility for automating common tasks (e.g., exporting logs, running workflows headless).
  - `Divooka.Web.Dashboard.exe`: A web-based dashboard front end for monitoring and controlling Divooka servers.
- **Libraries**
  - Contains all required assemblies for runtime, including Divooka.* DLLs and any third-party dependencies (e.g., JSON serializers, HTTP clients).
- **Examples**
  - A collection of sample workflow files - `.dvk` scripts that demonstrate use cases such as QR code generation or AI-powered image creation. Use these as starting points to learn how nodes are connected in real scenarios.
- **Licenses**
  - One license file per third-party component, detailing usage terms and attributions.
- **Plugins**
  - A directory where you can drop additional plugin DLLs. On startup, Divooka will scan this folder and load any compatible plugins to extend functionality.
- **UserManual**
  - Contains two subfolders: **BasicUsage** (guides for end users) and **Development** (documentation for developers writing plugins or extending Divooka).

By organizing the distribution this way, we keep each component modular. If you only need the core graph editor, focus on **Core Application** and **Engine**. If you want to build custom nodes, examine the **Libraries** and **Plugins** directories. If you simply want to see how workflows are structured, open the `.dvk` files in **Examples**.

1.4 Interface Tour

The Divooka graph editor presents a deceptively simple interface, hiding complex capabilities within nodes and toolboxes. Figure 1.1 shows an overview of Neo's interface, which is representative of all three flavors.

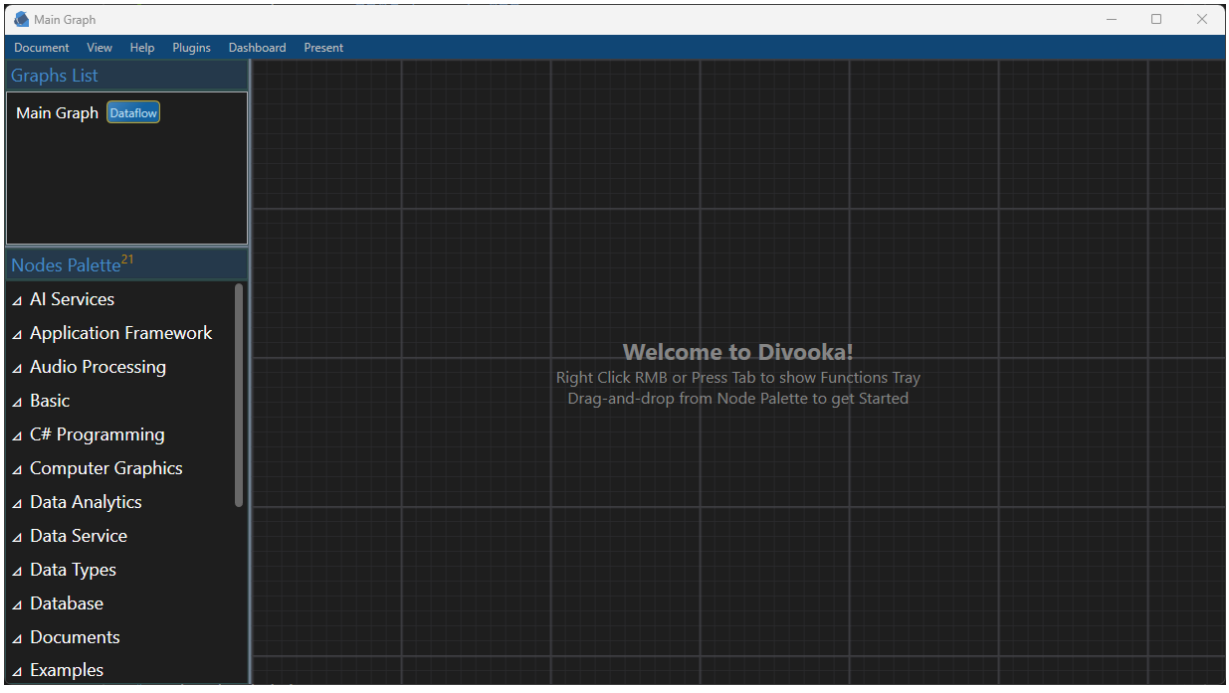


Fig. 1.1 Interface Overview

At first glance, you can identify four main regions:

1. **Main Menu** (File, Edit, View, Tools, Help) – provides global commands such as opening or saving workflows, configuring settings, and accessing documentation.
2. **Graphs List** (sidebar) – displays all open workflows and allows you to switch between multiple tabs or graphs.
3. **Toolbox Panel** (usually docked on the left or right) – a tree view of all installed toolboxes, categories, and functions. Drag a function from here onto the canvas to create a new node.
4. **Main Canvas** – the primary workspace where you arrange and connect nodes to build programs. You can zoom, pan, and select multiple nodes to move or delete them in bulk.

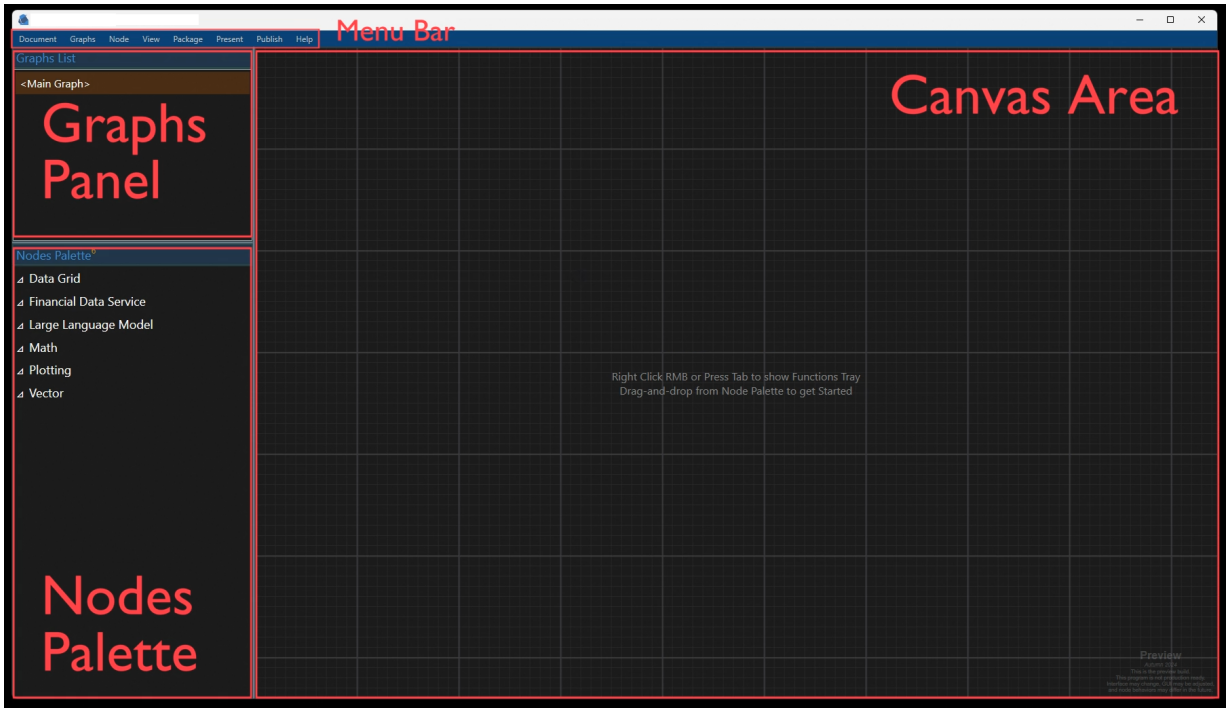


Fig. 1.2 GUI Menus

As you interact with nodes, additional windows may pop up to help you inspect intermediate data or configure settings. Of particular importance is the **Preview Window**, which shows intermediate results from nodes that produce visual or textual output. For example, when you run an image-generating node, the Preview Window can display the image it has produced.

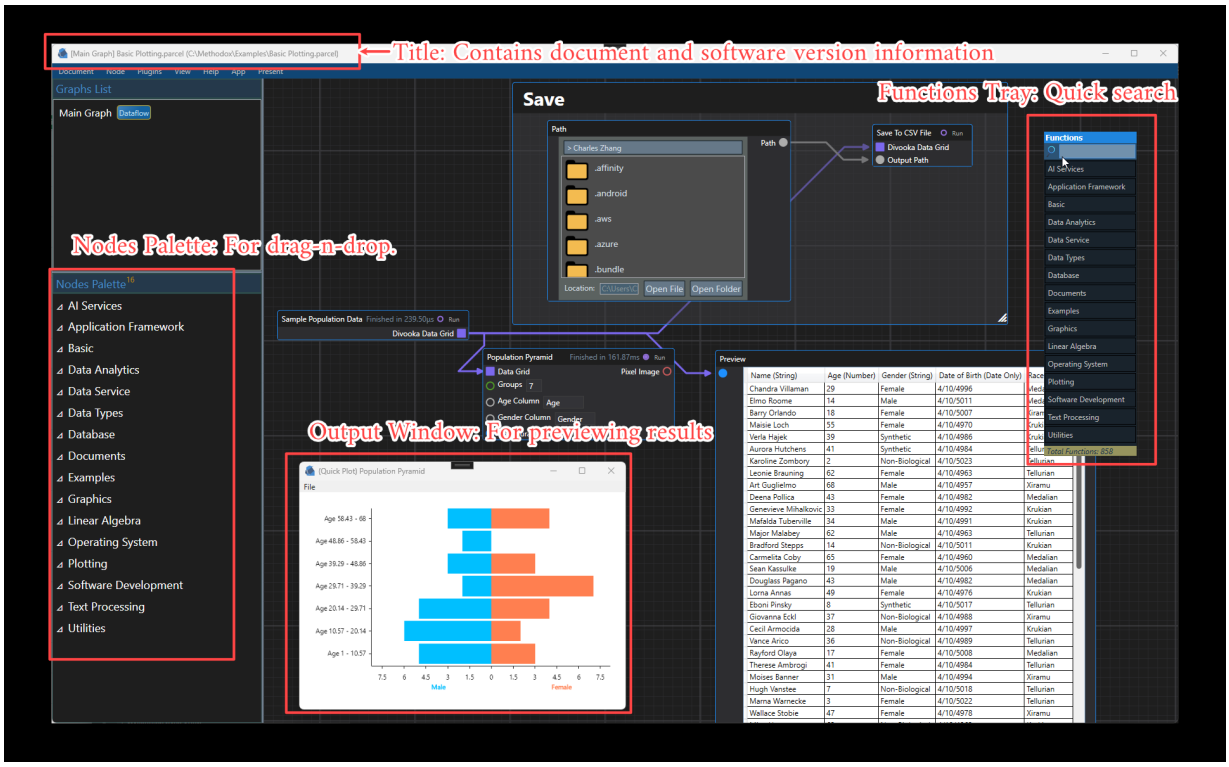


Fig. 1.3 Preview Windows and Popups

Learn to recognize these key areas - menu, graphs list, toolbox, canvas, and pop-up windows - so you can navigate the editor efficiently. In practice, you will spend most of your time arranging nodes on the canvas and occasionally opening the Preview Window to inspect data or images.

1.5 Structure of Nodes

Nodes are the foundational building blocks of Divooka workflows. Each node represents a single function or operation and consists of three main parts: **inputs**, **outputs**, and the **node surface**.

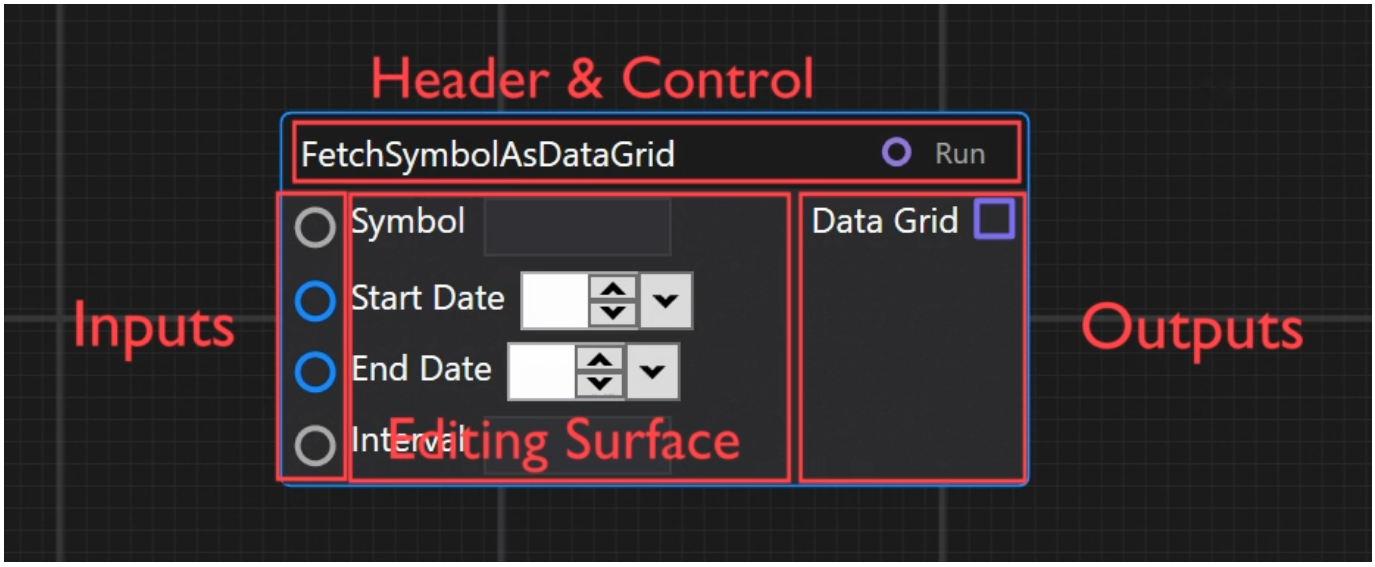


Fig. 1.4 Node Components

- **Inputs**: Represent the data or signals that the node consumes. Each input port typically has a label for the input name, and a symbol indicating the expected data type (e.g., string, number, image, JSON). You may hover over the port to see type name.
- **Outputs**: Represent the data or signals that the node generates. Downstream nodes can connect to these outputs to receive the node's results.

- **Node Surface**: The body of the node, where you can configure parameters or view status information. For example, a **Print Line** node's surface contains a text box where you enter the string to print, whereas a **Generate Image** node might include dropdowns for choosing resolution and style presets.

Programs in Divooka are simply graphs of nodes connected by wires. When one node's output connects to another node's input, it establishes a data flow: the first node must run and produce its output before the downstream node can execute. In the example below, an upstream node named **Load Data** feeds its output into the **Filter Rows** node, which then passes filtered results into a **Chart** node.

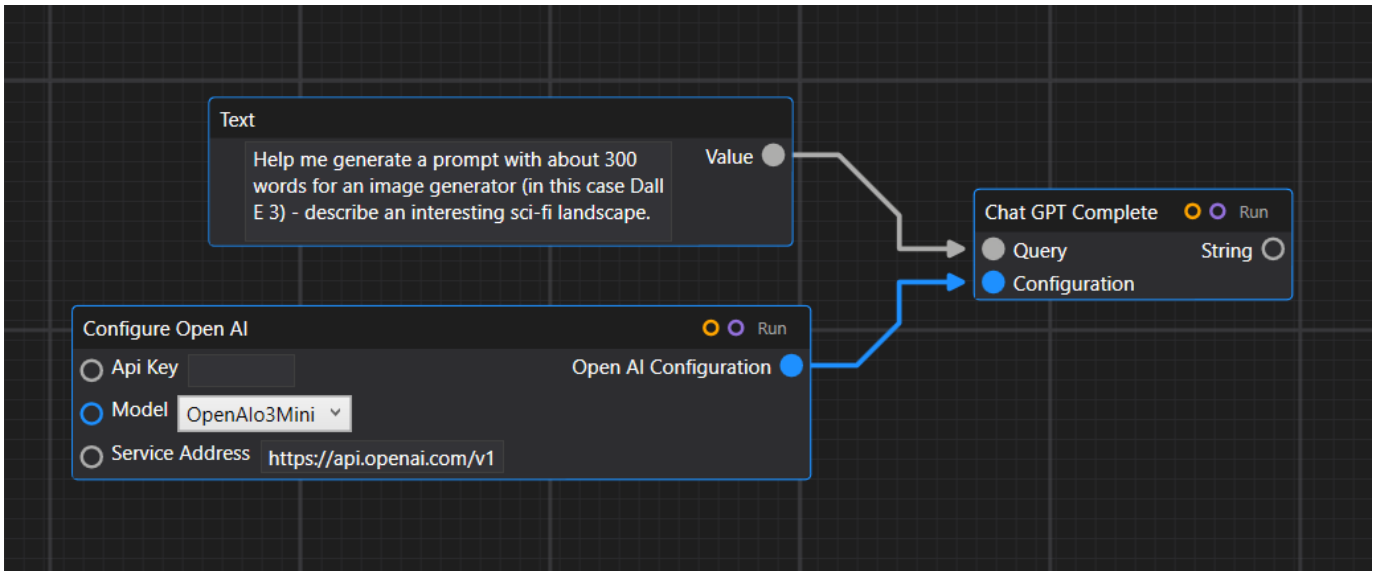


Fig. 1.5 Node Connections form Functional Programs

Some nodes have specialized surface controls. For instance, the **Inputs** node allows you to define multiple input parameters at once - clicking the "Add" button on the node surface creates new input fields dynamically. Below are examples of nodes with custom controls that let you adjust settings without opening a separate dialog.

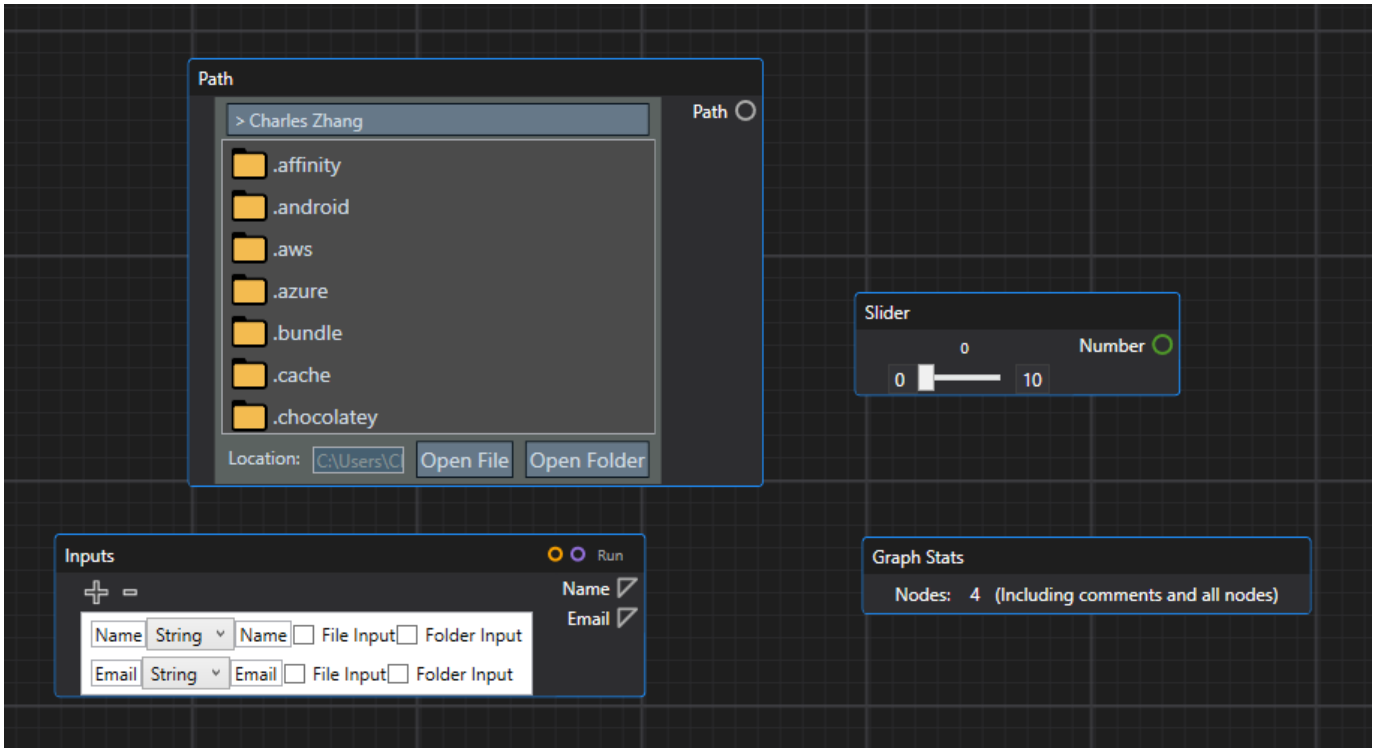


Fig. 1.6 Some Nodes with Specialized Surface Controls

When you connect nodes, the Divooka engine automatically determines the evaluation order. It starts with nodes that have no incoming wires (sources), evaluates them, then moves downstream. If a node's inputs are not yet ready (because

its upstream nodes have not produced outputs), Divooka waits until those nodes finish. This "lazy evaluation until dependencies are satisfied" model ensures that workflows run efficiently without redundant computations.

1.6 Your First Divooka Program

1.6.1 Hello World

As with most programming environments, our first Divooka example will print **"Hello World!"** to the console. Follow these steps:

1. **Open the Functions Tray**: Right-click on an empty spot in the canvas or press the `Tab` key. A searchable menu appears, listing all available functions organized by toolbox.
2. **Search for "Print Line"**: Start typing `Print` and select **Print Line** from the **Basic** toolbox.

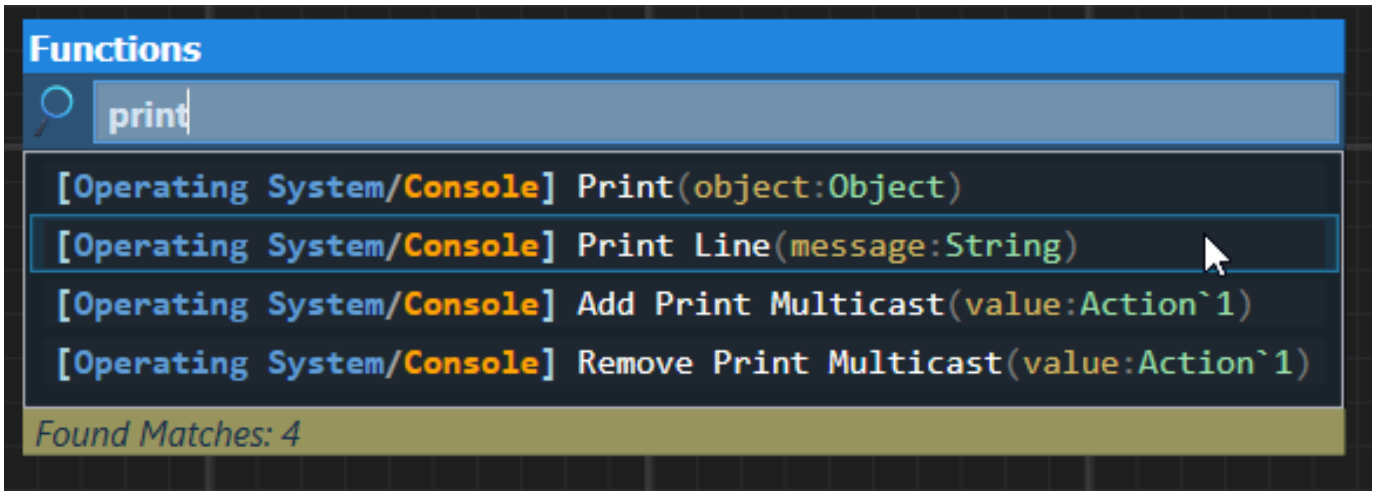


Fig. 1.7 Locating the Print Line Tool in Functions Tray

3. **Enter "Hello World!"**: On the **Print Line** node surface, click the text box labeled `Message`, and type in `Hello World!`.

Fig. 1.8 Enter "Hello World!" as Node Input

4. **Run the Node**: Click the small **Run** button on the top-right corner of the **Print Line** node. Divooka evaluates the node immediately and outputs "Hello World!" to the console - it's shown in the bottom right corner of the graph editor.

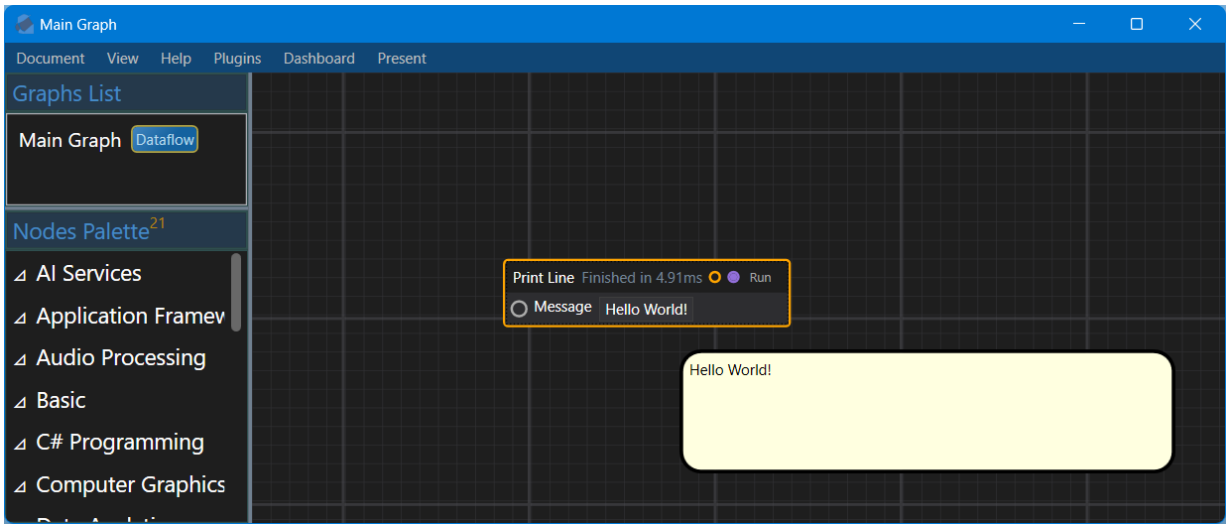


Fig. 1.9 Execute to Print

This simple example demonstrates how straightforward it is to get started: search for functions, configure node parameters, and run. Under the hood, Divooka parsed your workflow, found that the **Print Line** node had no upstream dependencies, executed it, and displayed the result.

1.6.2 A More Practical Example: Generative AI

"Hello World" is fine for a first program, but Divooka shines when you combine multiple nodes to accomplish real-world tasks. In this section, we will build a **prompt enhancer and image generator** that uses OpenAI services. Specifically, you will enter a short prompt, have ChatGPT expand that prompt into a detailed description, and then generate an image based on the detailed prompt. For simplicity, we will assume you have a **Methodox One** subscription, which includes built-in credentials for OpenAI and ComfyUI. If you prefer to run ComfyUI locally, adjust the **Generate Image** node to use your self-hosted endpoint.

1. **Open the Functions Tray** and search for **Chat GPT Complete** in the **OpenAI** toolbox. Drop that node on the canvas.

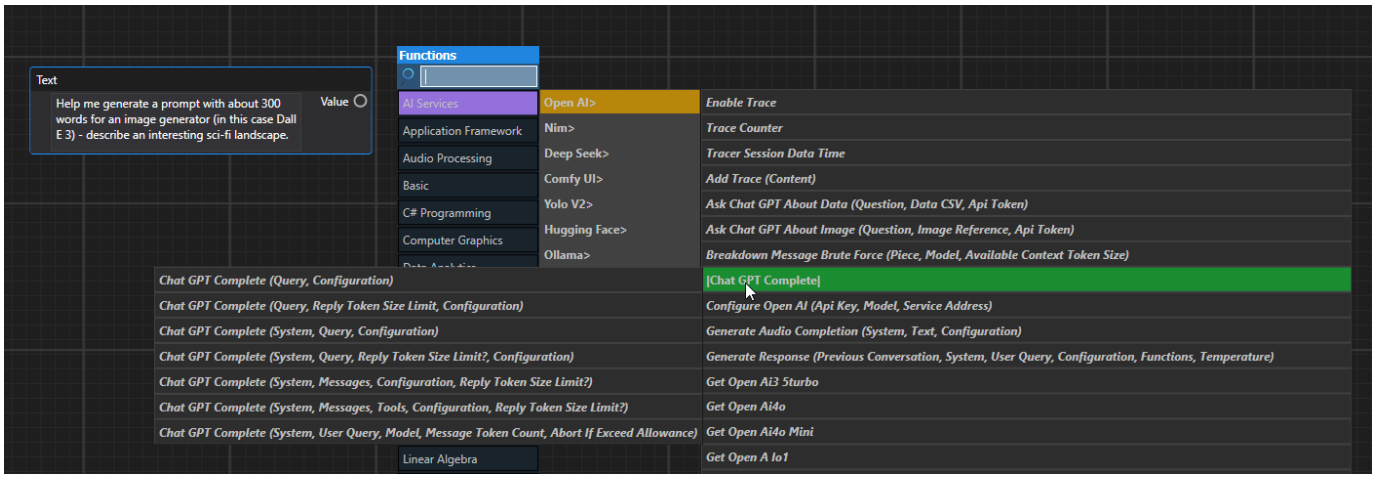


Fig. 1.10 Picking Chat GPT Complete from Toolboxes

2. **Add a "Generate Image" Node**: Still in the **OpenAI** toolbox, search for **Generate Image** (or **Simple Generate Image** under **ComfyUI** if you have ComfyUI configured). Drag it onto the canvas.

3. **Add Two Preview Nodes**: From the **Basic** toolbox, find **Preview** and drop it twice - one to preview the text output of ChatGPT and another to preview the generated image.

4. **Configure Node Connections**:

- Connect the **output** of **Chat GPT Complete** (parameter called `completion`) to the **input** of the first **Preview** node (to show the expanded text).
- Also connect the same `completion` output to the **prompt** input of **Generate Image**.
- Connect the **result** output of **Generate Image** to the **input** of the second **Preview** node (to show the image).

5. **Provide Your OpenAI Key**: Click on the **Chat GPT Complete** node and **Generate Image** node, then paste your OpenAI API key into the configuration field. You will see a placeholder like `sk-xxxxxxxxx...`. If you have a Methodox One subscription, this field may already be prefilled.

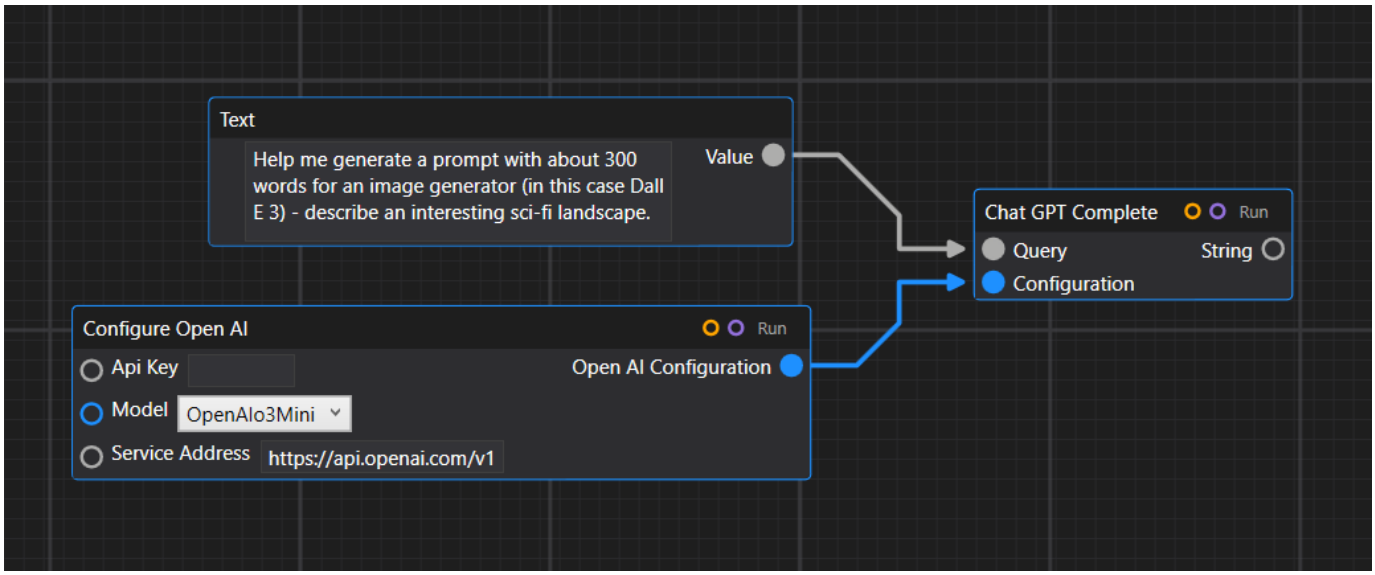


Fig. 1.11 Configure OpenAI Nodes

7. **Run the Workflow**: Click **Run All** (or select all nodes and press **F5**) so that Divooka evaluates each node in the correct order. First, **Chat GPT Complete** queries OpenAI to expand your prompt; that output is displayed in the first Preview. Then **Generate Image** uses the expanded prompt to create an image, which appears in the second Preview.

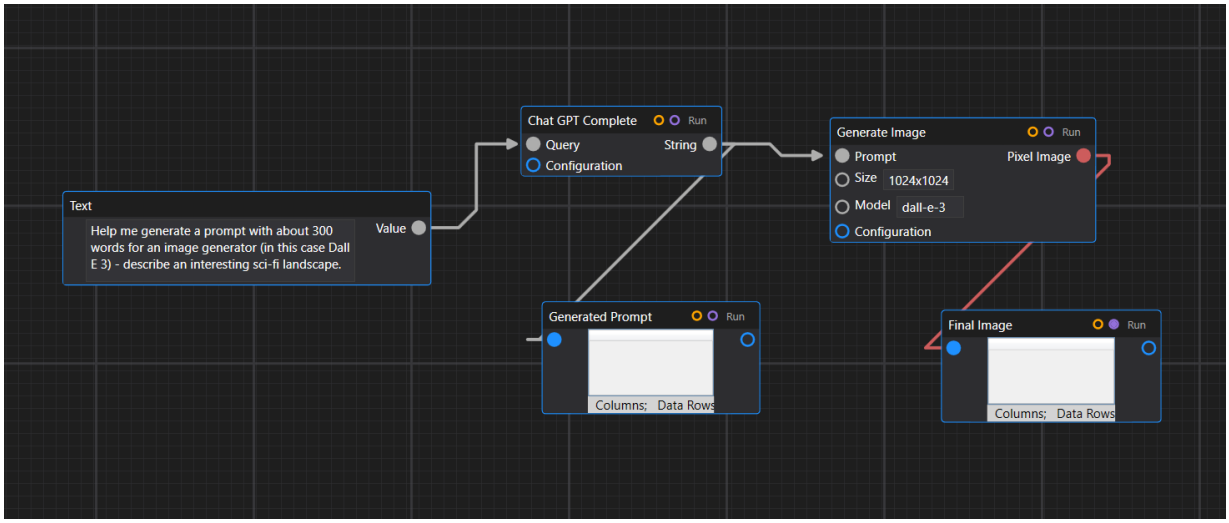


Fig. 1.12 A Practical First Program

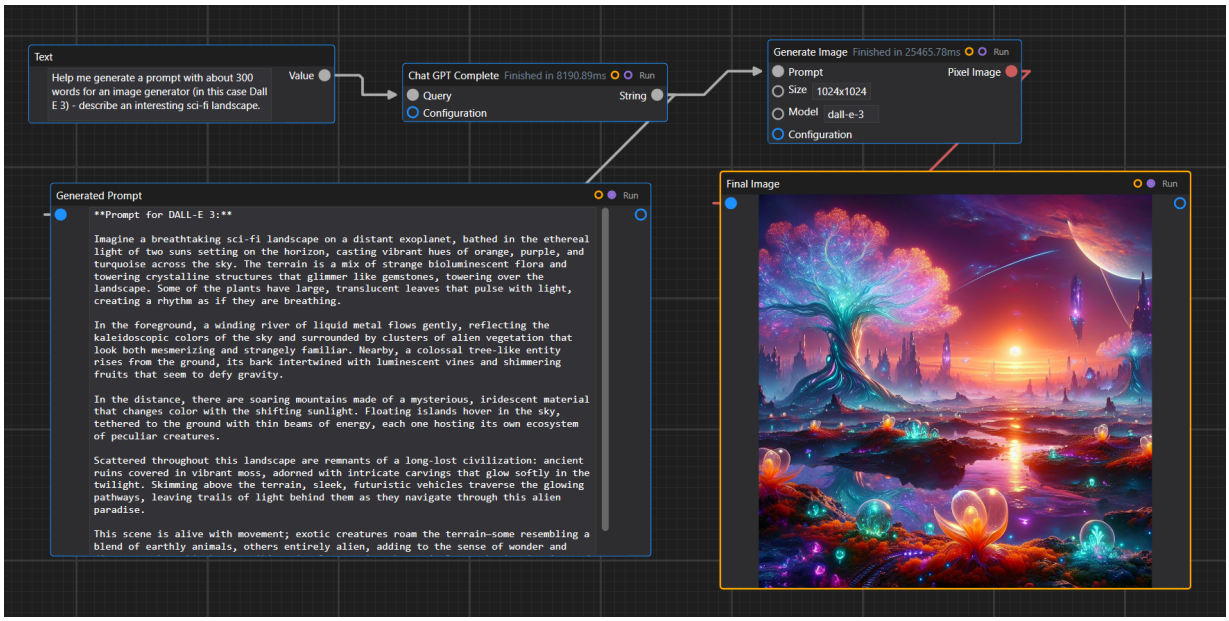


Fig. 1.13 Generative AI Result

1.6.3 What Just Happened?

- **Toolboxes**: Divooka organizes its thousands of functions under a hierarchy - Toolbox → Category → Function. For example, the **OpenAI** toolbox contains ChatGPT and image generation nodes, while the **ComfyUI** toolbox contains alternative generative AI nodes. If you install Divooka Compute, you can download additional toolboxes from official or third-party sources and share custom toolboxes among team members.

- **Node Parameters**: Each node's parameters (e.g., API key, prompt text, style presets) are displayed on the node's surface or in a properties pane. You can switch languages or locales, and the node's labels will adjust accordingly. Below is an example of the **Chat GPT Complete** node with Chinese labels:

(Node in Chinese)

- **Execution Flow**: When you press **Run**, Divooka analyzes the graph and constructs a dependency graph. It starts by evaluating nodes with no inputs (or inputs already satisfied). In this case, **Chat GPT Complete** has no upstream data dependencies except your API key, so it runs first. Once it returns text, that output feeds into the **Generate Image** node, which then sends the final image data to the second **Preview** node. Throughout execution, Divooka displays status icons (e.g., spinning gears, check marks) on each node to indicate whether it is waiting, running, or finished.

At this point, you have seen how easy it is to combine multiple services - text generation and image synthesis - into one coherent workflow without writing a single line of code. This modular, node-based approach allows you to focus on the logic and data flow rather than syntax. As you progress through Divooka's toolboxes, you'll find nodes for data import/export, conditional logic, loops, charting, API calls, and much more.

1.7 Summary

In this chapter, we introduced the core concepts and tools you need to start building workflows with Divooka. We covered the three official graph editors - Neo, Gospel, and Morpheus - and explained how they share a consistent interface despite differing in appearance. We walked through installation options, from the quick "Try It Online" environment to local installers, zip packages, and future source compilations, as well as Docker and mobile versions. We then explored the **Divooka Explore** distribution and its folder structure, showing how the core application, supporting tools, libraries, examples, and documentation are organized. Next, we took an in-depth tour of the interface, identifying the main menu, graphs list, toolbox panel, canvas, and preview windows so you can navigate the editor effectively. We explained the anatomy of nodes - inputs, outputs, and node surfaces - and how connecting nodes defines program flow.

Finally, we demonstrated two hands-on examples: a simple **Hello World** program and a more practical Generative AI workflow that uses ChatGPT and image generation. By the end of this chapter, you should have a solid understanding of Divooka's architecture, know how to install and launch the software, recognize the key interface components, and feel confident creating and running your first workflows. This foundation will prepare you for deeper dives into data processing, conditional logic, custom nodes, and advanced toolboxes in the chapters to come.

Chapter 2. Basic Concepts

> Through the translation of natural phenomena into structured data, humanity employs computation to craft algorithms that mirror reality's complexity, unleashing new horizons of engineered behavior.

2.1 Data and Their Types

(screenshot of an innocent looking node - string and number, e.g. range) (screenshot of an another node with different outputs - image and audio)

Nodes inputs can generally only take data with corresponding types. This is mostly true - though we will see exceptions.

2.2 Common Data Types

Learning any programming language fundamentally means understanding "what kinds of data" you can work with and "how" you manipulate them. In Divooka, every piece of data flowing through your graphs or stored in variables has a well-defined type. Because Divooka is a **strongly typed** language, each node's input and output ports expect values of a specific type - mismatched connections will be flagged immediately. In this section, we introduce the most basic and commonly used data types in Divooka, explain how to create constant values for each using the *Primitive* nodes, and describe their typical usage patterns. Wherever appropriate, we also point out subtle language features - especially around arrays - that let you work more flexibly with collections of data.

2.2.1 Number

Definition: The Number type in Divooka represents a numeric value - by default, a double-precision floating-point number. Internally, every Number is stored as a 64-bit IEEE-754 value, allowing both integers and real numbers (e.g., `42`, `3.14159`, `-0.5`).

Creating Number Constants: To inject a literal numeric value into your graph, drag out a **Number Primitive** node from the toolbox. Double-click (or select the node) to open its property editor, then type in any valid numeric literal. Once placed, this node exposes a single output port of type *Number*, which you can wire into any node input that accepts a Number.

Fig x.x (Insert a screenshot of a Number Primitive node showing "Value: 123.45")

Common Operations and Usage:

- **Arithmetic Nodes:** Nodes such as **Add**, **Subtract**, **Multiply**, and **Divide** all expect two (or more) Number inputs and emit a Number output.
- **Comparison Nodes:** Nodes like **Greater Than** or **Equal To** compare two Numbers and emit a Boolean.
- **Math Functions:** Nodes such as **Sin**, **Cos**, **Power**, or **Round** operate on Number inputs and produce Number outputs.

By placing a Number Primitive node and wiring it into a math node, you form a tiny expression graph. For example, a graph that multiplies `2.0` by `π` might look like:

```
[Number (2.0)] ──▶ [Multiply] ──▶ (Result)
            [Number (3.14159)] ──▶ ┘
```

Because Divooka infers execution order from data dependencies, when either Number Primitive node's value changes, the Multiply node re-evaluates automatically.

2.2.2 String, Text, Password, and Path

Underlying Type: Although these four primitives all carry textual information, Divooka treats them as distinct types at the GUI level to improve developer ergonomics. Internally, however, they all serialize to a standard .NET `string`. Each primitive node exposes the same "string" data under the hood but presents a different editor widget:

1. **String Primitive**
    - **Editor Style:** Single-line text box.
    - **Use Case:** Short labels, single-word identifiers, tags.
2. **Text Primitive**
    - **Editor Style:** Multiline text area (scrollable).
    - **Use Case:** Larger paragraphs of text (e.g., descriptions, multi-line user messages).
3. **Password Primitive**
    - **Editor Style:** Single-line, but masks characters (e.g., "••••••").
    - **Use Case:** Storing or passing sensitive text (passwords, tokens).
    - **Note:** When displaying Password output in the graph, Divooka will still show a masked value. If you feed a Password into a node that expects a plain String, it is implicitly cast to a String (but the runtime nodal viewer will not reveal the raw characters).
4. **Path Primitive**
    - **Editor Style:** Provides a file-system picker for files and folders.
    - **Use Case:** Selecting disk-based resources (e.g., images, CSV files, configuration directories).
    - **Note:** The Path node includes "Open (Browse File)" and "Open Folder (Browser Folder)" buttons; once a path is chosen, its value appears as a string, e.g. `C:\Users\Alice\Documents\data.csv`.

Fig x.x (Insert adjacent screenshots of String, Text, Password, and Path Primitives side by side.)

Typical Workflow with Text Primitives:

- **Concatenation:** Use the **String Concat** node - this node expects one input of type `object[]` (i.e., an array of objects) and concatenates each element's `ToString()` representation. Because of Divooka's *array coercion* feature (see §2.2.7 below), you can connect multiple single-object inputs directly into the Concat node, or you can assemble an array via a literal Array Primitive and connect that.
- **Substring/Split/Replace:** Divooka provides dedicated nodes for common string-processing tasks: **Substring**, **Split**, **Replace**, **Trim**, and so on. Each node declares its string-typed inputs explicitly.

2.2.3 DateTime

Definition: The DateTime type represents an instant on the global timeline (year, month, day, hour, minute, second, and millisecond). Divooka stores DateTime values internally as a standard .NET `DateTime` object, which can represent dates ranging from January 1, 0001 CE through December 31, 9999 CE.

Creating DateTime Constants:

- Drag a **DateTime Primitive** node onto the canvas.
- Click the embedded calendar/time picker - Divooka's UI shows a calendar control plus time selectors. Select the desired date and time, then click "OK." The node's output port (type *DateTime*) now emits that exact instant.

Fig x.x (Insert a DateTime Primitive with "2025-06-01 14:30:00" selected.)

Common Usage Patterns:

- **Arithmetic:** Use nodes such as **Add Days**, **Add Hours**, or **Subtract** (two DateTimes → a `TimeSpan`).
- **Formatting:** The **FormatDate** node will convert a DateTime into a String based on a format specifier (e.g., `"yyyy-MM-dd HH:mm"`).
- **Comparison:** Nodes like **Before?** or **After?** take two DateTime inputs and emit a Boolean.
- **Split Components:** Nodes such as **DatePart** allow you to extract the year, month, or day as a Number.

Because DateTime flows as a strongly typed value, you cannot mistakenly wire a DateTime Primitive into a Number-only port. If you need a numeric representation (for example, a Unix timestamp), you must chain the DateTime node into a **ToUnixTimestamp** node, which emits a Number.

2.2.4 Color

Definition: Color represents an RGBA (Red, Green, Blue, Alpha) quadruple. Internally, Divooka stores Color values as a .NET `Color` struct with 8 bits per channel (0–255).

Creating Color Constants:

- Place a **Color Primitive** node.
- Click the swatch to open a color-wheel plus opacity slider. Choose any color or type hex/RGB values directly. Once confirmed, the node's output port (type *Color*) emits a four-byte color value.

Fig x.x (Insert a Color Primitive showing a semi-transparent teal, e.g., `#00808080`.)

Common Operations:

- **Blend:** A **BlendColor** node takes two Color inputs plus a Number (0.0–1.0) factor and outputs a new Color.
- **Invert:** An **InvertColor** node flips each channel (e.g., `R → 255−R`, etc.).
- **Component Extraction:** Use **ColorToChannels** to split a Color into four separate Number outputs (R, G, B, A).
- **Convert to Hex:** The **ColorToHex** node formats a Color as a String (`"#RRGGBBAA"`).

Color values are essential for any UI-related, graphics, or visualization tasks. Because Divooka is strongly typed, a Color output cannot drive a Number input without passing through a specialized conversion node first.

2.2.5 Boolean

Definition: A Boolean in Divooka is a simple `true` or `false` value, corresponding to .NET's `bool` type. It is typically used for conditional checks, toggles, and on/off flags.

Creating Boolean Constants:

- Drag out a **Boolean Primitive** node.
- The node presents a checkbox (or a toggle) to pick either `true` or `false`. The output port emits a Boolean.

Fig x.x (Insert a Boolean Primitive showing "Value: true".)

Common Logic and Operations:

- **Logical Operators:** The **And**, **Or**, and **Not** nodes each take Boolean inputs and emit a Boolean result.
- **Comparison Nodes:** When comparing Numbers (e.g., `x > y`) or Strings (e.g., `Text1 == Text2`), Divooka automatically outputs a Boolean.
- **Conditional Nodes in Dataflow:** A **Select** (or **Ternary**) node takes a Boolean condition plus two alternative values (of the same type) and outputs one of them based on the Boolean.

Because Boolean is a distinct primitive type, attempts to connect a Number into a Boolean-only port (or vice versa) will result in a compile-time type error, ensuring your logic remains semantically correct.

2.2.6 Data Grid and Data Column

Definition:

- A **Data Column** represents a single column of tabular data (e.g., a list of Numbers, Strings, or Dates). Internally it corresponds to a generic `IEnumerable<T>` for some element type `T`.
- A **Data Grid** is a two-dimensional table, analogous to a `DataTable` or a collection of `DataColumn` objects. It can hold multiple rows and columns of mixed types.

Creating Data Grid/Column Constants:

- Use the **Data Column Primitive** to manually type in or paste a list of values (comma-separated) for one column. You must select the column's element type (Number, String, DateTime, etc.) before entry. Once entered, the node emits a `DataColumn<T>` object.
- Use the **Data Grid Primitive** to define a small inline table: first add column headers (with explicit types), then add rows one by one. The grid node's output port emits a `DataGrid` object.

Fig x.x (Insert a Data Grid Primitive showing a 3×2 table: columns "Name (String), Age (Number)", rows ["Alice", 30], ["Bob", 25], ["Cara", 28].)

Typical Operations:

- **Select Columns:** A **GetColumn** node takes a DataGrid plus a column name (`string`) and emits the corresponding DataColumn.
- **Filter Rows:** A **FilterRows** node takes a DataGrid plus a predicate (a lambda function that returns Boolean for each row) and emits a smaller DataGrid.
- **Join/Merge:** Combine two DataGrids using keys by using **JoinTables**.

Because Data Grid and Data Column are reference types wrapping collections, they interoperate with array and generic functions (see §2.2.7), enabling powerful tabular data transformations in a mostly visual manner.

2.2.7 Arrays of Data and Coercion

(Example screenshot)

Definition: An array is a collection of elements of the *same type*. In Divooka, every array is represented internally as an `IEnumerable<T>` (e.g., `IEnumerable<Number>` or `IEnumerable<String>`), but at design time, you will see ports labeled as `T[]` or `object[]` (for untyped or polymorphic arrays).

Creating Array Constants:

- Drag an **Array Primitive** node (choose the element type, e.g., `Number[]`, `String[]`, `Object[]`).
- In the property editor, either manually type comma-separated literals (e.g., `1, 2, 3, 4`) or connect multiple individual Element Primitives to the Array node's "Elements" collection input.

Fig x.x (Insert an Array Primitive of type `Number[]` with values `[5, 10, 15]`.)

Array Coercion (Type Flexibility): Because Divooka is strongly typed, you might expect that a node whose input is declared as `Type[]` would only accept a literal array. However, Divooka provides **array coercion**, meaning:

If an input port expects `Type[]`, you may either:

1. Connect a `Type[]` array directly, or
2. Connect zero or more individual `Type`-typed wires.

At runtime, Divooka "collects" all incoming `Type` wires into a single array. If you connect only one wire carrying a `Type` value, Divooka automatically wraps it into a singleton array `{ T }`. If you connect no wires, the node sees an empty array

(equivalent to `{ }`).

Example: String Concat with array coercion The **String Concat** node is defined as:

```
Input: object[] items
Output: String result
```

You can wire either:

- A literal `object[]` (via an Array Primitive whose element-type is Object), or
- Several separate connections of primitive values (e.g., a Number node, a String node, a Boolean node) - Divooka will box each into an `object` and build an array for you.

```
[Number (42)] ──▶ ┐
[Text (" apples and ")] ──▶ │
[Number (100)] ──▶ │
   (no Array Primitive)   [Concat] ──▶ "42 apples and 100"
```

In this example, although Concat declares an `object[]` input, you do not need a separate Array node - Divooka's coercion collects the three wires (Box 42 → object, " apples and " → object, Box 100 → object) into an `object[]` at runtime.

Arrays vs. Generic Functions (Apply, Select, etc.) Generic data-manipulation nodes (e.g., **Apply**, **Where**, **GroupBy**) originate from LINQ's `IEnumerable<T>` methods. These nodes are implemented as generics. When you connect the first input (an `IEnumerable<U>`) into an Apply or Select node, Divooka's graph engine "binds" the generic type parameter `T` to `U`. After binding, the node expects a `Func<U,R>` for the selector lambda, and its output is `IEnumerable<R>`. Critically, the array-coercion rule does **not** apply to these generic nodes: you cannot feed a single `U` primitive directly into an `IEnumerable<U>` port on Apply - first you must wrap it in an Array Primitive (or have another node produce a collection).

Example (Correct):

```
[Array Primitive (Number[] {1, 2, 3})] ──▶ [Apply (generic Number→Number)] ──▶ (Result:
Number[])
```

Example (Incorrect):

```
[Number (1)] ──▶ [Apply (generic Number→Number)]  ← (This will error, because Apply
expects an IEnumerable<Number>, not a single Number)
```

Divooka is strongly typed. Incompatible value type connections are not allowed. Every time you make a connection, Divooka verifies that the **output type** of Node A exactly matches the **input type** of Node B (with the only exception being arrays of primitives, as described above). If you attempt to connect a Boolean output into a Number input, the graph will refuse to connect until you insert a conversion node (e.g., **BoolToNumber**). When opening a new graph, the editor will also check whether is invalid connections in the source file.

Divooka provides a suite of **ToX** nodes for casting between compatible types. For example, **ToString** (any input → String), **ToNumber** (String → Number, if the text is parseable), and **ToDateTime** (String → DateTime). These conversion nodes let

you cross "type boundaries" when necessary.

2.3 Dataflow Context vs Procedural Context

In Divooka, you can structure your programs in two complementary ways, each with its own usages and visual style. The **Dataflow Context** is designed around the idea of data "flowing" through a network of nodes, where each node represents an operation or transformation. In contrast, the **Procedural Context** mimics the familiar sequential model of traditional text-based programming, with a linear sequence of steps, loops, and conditional branches. Although both contexts can ultimately accomplish similar tasks, understanding their respective strengths and limitations will help you choose the appropriate paradigm for a given problem.

2.3.1 Dataflow Context

The Dataflow Context emphasizes declarative wiring between components. In this paradigm, you place nodes on a canvas and connect their output ports to the input ports of other nodes. During execution, the node evaluates its outputs based on currently available inputs, and that new values propagates downstream to any connected nodes. This model makes it exceedingly easy to visualize how data is transformed step by step.

Core Concepts:

- **Nodes as Functions:** Each node encapsulates a simple function - such as mathematical operations, data formatting, or basic I/O.
- **Wires as Dependencies:** A wire indicates that one node depends on the output of another. Whenever the source node's output changes, the destination node is triggered to recompute.
- **Implicit Execution Order:** Because execution is driven by changes in data, you do not explicitly specify "which node runs first." Instead, Divooka infers the evaluation order based on the topology of your graph.

Conditional Logic and Subtleties

While simple "if/then/else" constructs are available (for example, through a dedicated "Select" node), there are a few subtleties to keep in mind:

1. **Evaluation of Both Branches:** Divooka evaluates both "true" and "false" branches of a conditional node before choosing which output to propagate. This means any side effects in the non-selected branch could still occur unless you take care to wrap them appropriately.
2. **No Native Loops - Only Recursion or Specialized Nodes:** Traditional loops (for/while) do not exist directly in a dataflow canvas. Instead, you use specialized "Apply" nodes that internally manage repetition.

When to Use Dataflow:

- **Visual Clarity:** If your task is fundamentally about passing data through a series of straightforward transformations - such as signal processing, spreadsheet-style calculations, or pipelined transformations - a dataflow graph provides excellent visual clarity.
- **Reactive Programming:** For applications where inputs may change at unpredictable times (user interactions, sensor data, streaming inputs), the dataflow paradigm makes it easy to propagate updates automatically without manually managing execution order.

2.3.2 Procedural Context

The Procedural Context in Divooka will feel familiar to anyone who has written imperative code in languages like C, Java, or Python. Here, you explicitly lay out the sequence of operations, control flow statements, and loop constructs, all within a flowchart-style diagram. Each block (or "step") represents a discrete instruction, and arrows indicate the exact order in which those blocks should execute.

Core Concepts:

- **Sequential Blocks:** Each block corresponds to a specific operation (e.g., variable assignment, arithmetic operation, function call). You place them on the canvas and draw arrows to indicate the order in which they run.
- **Control Flow Constructs:**
    - **Branching (If/Else):** A decision block evaluates a Boolean expression; based on the result, execution continues along one of two (or more) outgoing paths.
    - **Loops (For/While):** Specialized loop constructs allow you to repeat a sequence of blocks until a condition is met.
    - **Subroutines and Functions:** You can define custom events or use new graphs to encapsulate a sequence of blocks into a reusable "procedure" and invoke it from multiple places in your main flow.
- **Explicit Order of Execution:** Unlike Dataflow, you must explicitly connect blocks in the order you wish them to run; this gives you precise control over when and how each statement is executed.

Data Handling

- **Variables and State:** In a procedural context, you explicitly declare and manipulate variables. When one block writes to a variable, its value remains until another block changes it.
- **Passing Data Between Blocks:** Variables serve as the primary mechanism for passing information between different parts of the program. This is in contrast to Dataflow, where wires themselves carry values directly from one node to another.

When to Use Procedural:

- **Complex Control Logic:** If your algorithm requires nested loops, intricate conditional branching, or deeply sequential logic (e.g., initialization steps followed by iterative refinement), the procedural context makes the flow of control very explicit.
- **Algorithmic Familiarity:** For tasks like implementing sorting algorithms, classical state machines, or finite-step simulations, many programmers find the procedural diagram more intuitive because it mirrors the logic they would write in text form.

2.3.3 Comparing the Two Contexts

Aspect	Dataflow Context	Procedural Context
Execution Model	Reactive, driven by data dependencies	Explicit, driven by directed control flow
Branching and Loops	Conditional nodes; loops via feedback constructs	Standard if/else, for, while blocks
State Management	Minimally stateful; most nodes are pure functions	Explicit variables and state updates
Visualization Strengths	Excellent for pipelines and real-time data flow	Excellent for step-by-step algorithms
Best Suited For	Data transformation pipelines, streaming inputs	Procedural algorithms, tasks requiring order

In practice, Divooka allows you to mix and match: you might build a core data-transformation pipeline in Dataflow for clarity, then invoke a small Procedural subroutine whenever you need to perform a complex loop or branching sequence. By understanding both paradigms, you can choose the one that makes your logic easiest to read, maintain, and debug.

Below is the representative screenshot illustrating a simple Dataflow graph beside a comparable Procedural flowchart.

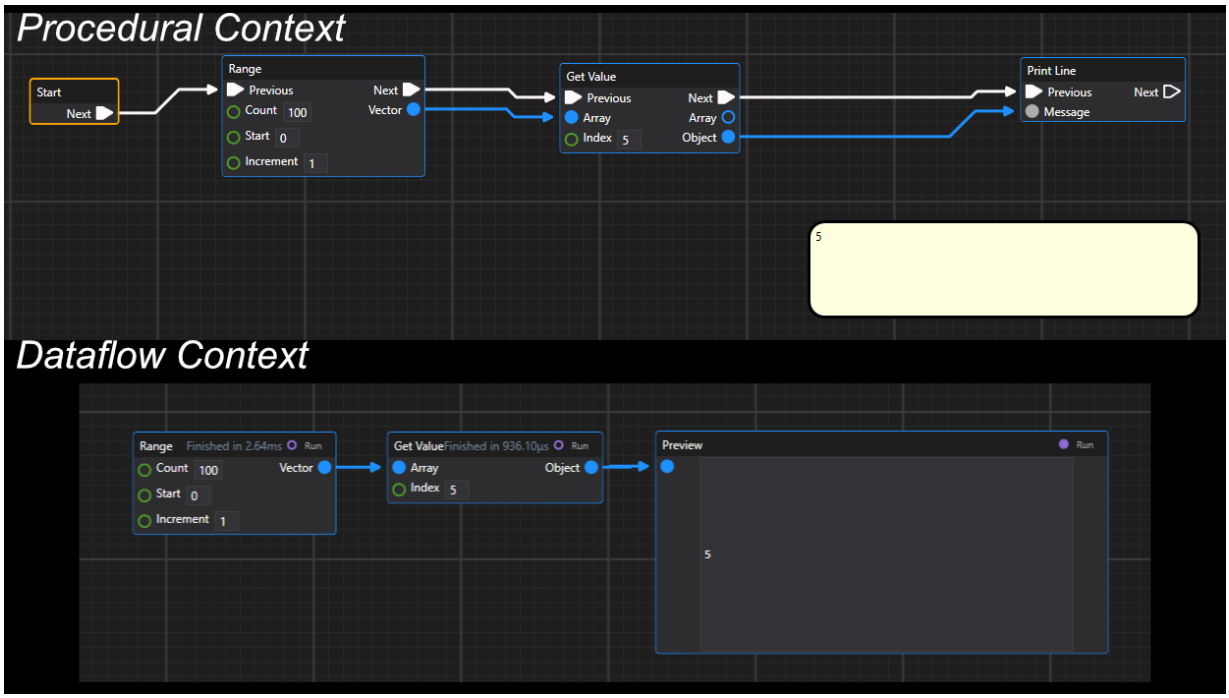


Fig. 2.1 Dataflow Context vs Procedural Context

2.4 Summary

Divooka's powerful combination of a **strongly typed** system and its dual paradigm (Dataflow and Procedural contexts) hinges on clearly understanding each data type. In this chapter, we covered:

1. **Number:** 64-bit floating-point values, created via the Number Primitive, used for arithmetic and comparisons.
2. **String/Text/Password/Path:** Four specialized Primitive nodes that share the same internal type (`string`), but differ in UI: single-line, multiline, masked entry, and file-system path selection, respectively.
3. **DateTime:** A calendar/time picker Primitive for representing instants on the timeline, used with specialized date/time arithmetic and formatting nodes.
4. **Color:** An RGBA picker Primitive, essential for any UI/graphics tasks, with blend, invert, and conversion utilities.
5. **Boolean:** A simple true/false Primitive, used for conditional logic and control-flow decisions.
6. **Data Grid/Data Column:** Tabular data structures for handling row/column collections, enabling data-analysis workflows.
7. **Arrays of Data:** Collections of homogeneous elements (`Type[]`) with special "array coercion," letting many functions accept either a full array or separate element wires.

By mastering these basic types and how Divooka enforces type consistency - yet still lets you work flexibly with arrays - you lay the groundwork for constructing correct, maintainable visual programs.

Chapter 3. Runtime Matters

In this chapter, we dive into how Divooka's Dataflow graphs actually run under the hood. Unlike procedural flowcharts, where each step follows the previous one in a clearly defined linear order, dataflow programs rely on dependencies between nodes to determine what executes and when. Because you never explicitly write "line 1, line 2, line 3" in a dataflow canvas, a few subtleties arise around activation, caching, and dependency ordering. In the sections below, we will explore:

1. **Dependency and Execution Order** – how the "Run" button and active status drive evaluation
2. **Active Nodes** – what it means to set a node to active, and how Divooka orders and evaluates active nodes
3. **Node Cache** – how enabling cache changes which nodes get re-evaluated and which results are reused

By the end of this chapter, you will understand exactly how to control execution in Divooka's Dataflow Context, avoid unwanted recomputation, and prevent hidden dependency pitfalls.

3.1 Dependency and Execution Order (Dataflow Context)

In a procedural, textual language, it's obvious which statement executes first: you wrote them in order. In Divooka's Dataflow Context, however, you do not specify an explicit "step 1, then step 2." Instead, execution is driven by **data dependencies**: whenever a node becomes "active," Divooka examines its inputs, finds all upstream nodes that feed it, and then evaluates those inputs in an order that respects dependency. Finally, it propagates computed values downstream.

3.1.1 The "Run" Button and Visual Cues

Every node in Divooka's canvas includes a small **Run** button at its top-right corner. Clicking this button serves two purposes:

1. **Activate the Node:** The node's purple indicator circle (to the right of the Run button) turns on, signaling that it is "active."
2. **Trigger Evaluation:** Divooka's runtime begins gathering all currently active nodes, sorting them by dependency, and then computing values in the necessary order.

Next to the Run button are two status circles:

- **Purple Circle (Active Status):** When filled, this means you have requested that this node and all its activated dependencies should be recomputed.
- **Orange Circle (Cached Status):** When filled, this means the node is in "cache mode" - the runtime will reuse its last output instead of recomputing it, even if it is active.

Visual Example:

Imagine a simple graph with three nodes:

```
[Primitive Number (5)] ──▶ [Number Multiply (by 2)] ──▶ [Preview]
```

If you click "Run" on **Preview**, the runtime sees it is active. It then discovers that "Preview" depends on "Double," which depends on the Number Primitive. All three become part of the dependency chain. Divooka first evaluates the Number Primitive (trivially, because it's a literal), then "Double," then finally "Preview." Once done, the graph shows the computed result (here, 10) at each node's output port.

3.1.2 Dependency Sorting

When Divooka begins a run, it must decide in what order to evaluate each node. In the current implementation, the runtime performs a **depth-first** traversal - starting from downstream active nodes, it recursively visits each upstream node until it reaches a literal (a Primitive) or a cached node. Once it hits a leaf, it bubbles back up, evaluating intermediate nodes as it returns. The result is that all dependencies are computed before any node that relies on them.

- **Leaf Nodes (Primitives):** Always run first, unless cached.
- **Intermediate Nodes:** Run only after all upstream inputs have been evaluated (or loaded from cache).
- **Downstream Nodes:** Run last, so their inputs are guaranteed to be up-to-date.

Subtlety to Note: Because the current algorithm is depth-first, it may visit one branch of the graph completely before exploring a parallel branch. In practice, this rarely matters for pure functions (nodes without side effects). However, if you place a node that writes to disk or logs externally as part of one branch, you may notice that it executes to completion before another branch begins, even if that other branch is "closer" to the root of your active node. In future versions, Divooka may switch to a topological sort or breadth-first approach, so you should not rely on depth-first ordering for correctness.

3.2 Active Node

When you click a node's **Run** button, the node gains "Active" status - meaning it and all of its dependencies will be evaluated on the next execution pass. However, activating a node is more nuanced than "just run this one node." In fact, Divooka's runtime follows these steps:

1. **Mark Node as Active:** The purple circle beside the Run button fills in.
2. **Collect All Active Nodes:** The runtime builds a set of every node in the current graph whose purple circle is filled (including the one you just clicked).
3. **Compute Dependency:** For each active node, Divooka finds all upstream nodes (recursively) that feed into it.
4. **Filter out Cached Nodes:** Any node you have toggled to "cached" will not be re-evaluated unless it has never run before. Instead, its previous output will be used.
5. **Sort by Dependency:** The runtime orders nodes so that upstream dependencies are computed before downstream dependents.
6. **Evaluate in Order:** Finally, Divooka evaluates each node in turn, using a depth-first approach under the hood. If a node's inputs all come from cached (and up-to-date) nodes, Divooka may skip re-evaluating it entirely (unless it's never run).

Because of this mechanism, clicking **"Run"** on a single node can cause large subgraphs to recompute - whenever you have multiple active nodes, Divooka will, in effect, merge their upstream dependencies into one big dependency graph for that run. Any node that is already active, even if you do not re-click its Run button, will also reevaluate (unless cached). This behavior ensures that the graph is always consistent: if Node A depends on Node B, and Node B's output has changed since the last evaluation, then Node A will automatically recompute the next time you run any active node downstream.

Example: Parallel Active Nodes

Suppose you have two separate pipelines in one graph:

```
[Number(3)] ──▶ [Square] ──▶ [ResultA]

[Number(4)] ──▶ [Square] ──▶ [ResultB]
```

You click "Run" on both **ResultA** and **ResultB**. Divooka's runtime builds the dependency tree for each:

- For **ResultA**, the upstream dependency is `[Number(3)] → [Square]`.
- For **ResultB**, the upstream dependency is `[Number(4)] → [Square]`.

Even though these are two disjoint subgraphs, both "Square" nodes (which may be the same re-used node or two instances, depending on your design) will run with their respective inputs. The runtime may choose to evaluate one branch entirely before the other, but because they share no dependencies, their order does not affect correctness.

3.3 Node Cache

Divooka lets you toggle **Cache** on any node. A cached node behaves as follows:

- **Does Not Recompute on Run:** If the node has run at least once in the past, Divooka will skip executing its function and simply return the last stored output value.
- **Still Allows "First Run":** If you set a node to cached before it has ever executed, the very first run will compute its value; after that, it becomes "frozen" until you toggle Cache off or otherwise invalidate it.
- **Filters Out Dependencies:** When the runtime is determining which nodes to evaluate, any cached node is treated as a "leaf" (even if it has upstream inputs). That means its upstream inputs will not run purely to service that cached node - unless those upstream nodes are active for some other reason.

In other words, once a node is cached:

1. It no longer appears in the "need to compute" set (except for the first time).
2. Any node that depends on it sees the cached value and does not force a re-evaluation of its upstream graph for that branch.
3. If you have downstream nodes active (and not cached), the propagation will "stop" at the cached node: it supplies its last result but does not pull new data.

Example: Caching a Compute-Intensive Node

Imagine a node called **GenerateLargeList** that runs an expensive database query or builds a huge in-memory structure. You connect it to several downstream analysis nodes. If you know that the source data does not change, you can click "Cache" on **GenerateLargeList**. Now, every time you click "Run" on one of the analysis nodes, Divooka will treat **GenerateLargeList** as a leaf - fetching the previously computed list without hitting the database again. Downstream nodes recompute, but the heavy work of regenerating that list is skipped.

Subtlety: Cached + Active

If you set a node to **active** and **cached** at the same time (i.e., both circles are filled), Divooka gives priority to caching. The node's Run button will still show that it is active, but it will not recompute on that run pass. Importantly, any upstream nodes that feed into this cached node will also be trimmed from the dependency graph - because Divooka knows it does not need fresh inputs.

Only nodes that are both **active** and **not cached** will truly execute in a run pass.

3.3.1 Cache Invalidation

The only ways to invalidate a cached node are:

- **Toggle Cache Off:** Click the orange circle to disable caching. The next time you run any active node that depends on it, it will recompute.
- **Manually "Clear Cache" Command:** Divooka's right-click context menu on any node offers a **Clear Cache** action, which forces that node (and only that node) to recompute on the next run, even if its orange circle remains filled.

Notice even if you have modified the input values or reconnected the input connectors of a node, if it has Cache toggle on, then it will not automatically recompute.

3.4 Practical Guidance and Examples

To tie these concepts together, let's look at a few real-world scenarios you might face while building interactive dataflow graphs:

Scenario A: Rapid Prototyping with Caching

You are designing a financial dashboard in Divooka:

1. **Step 1**: Create a **LoadCSV** node that reads a large CSV file containing months of transaction data.
2. **Step 2**: Connect **LoadCSV** to a **CalculateMovingAverage** node (which itself is relatively expensive over thousands of rows).
3. **Step 3**: Connect that to a **PlotChart** node, which renders a chart for you.

While you are experimenting with different chart styles, the underlying CSV data does not change. To avoid repeated CSV parsing and moving-average computation, you click "Cache" on **CalculateMovingAverage**. Now, every time you click "Run" on **PlotChart**, Divooka will use the cached moving-average results, redrawing only the chart. If you later switch to

a new data file, simply edit the **LoadCSV** node's path (or right-click and "Clear Cache" on **CalculateMovingAverage**), then run again - everything updates.

Scenario B: Avoiding Unnecessary Computation in Branching Graphs

Imagine a complex graph that splits into two parallel feature-extraction pipelines:

```
[LoadImage] ──▶ [Resize] ──▶ [ExtractFeaturesA] ──▶ [ClassifierA]
                    │
                    └──▶ [ExtractFeaturesB] ──▶ [ClassifierB]
```

You want to compare two classification models side by side. If you click "Run" on **ClassifierA**, Divooka will evaluate `LoadImage` → `Resize` → `ExtractFeaturesA` → `ClassifierA`. It will ignore the entire branch containing **ExtractFeaturesB** and **ClassifierB**, since they are not marked active. If later you mark **ClassifierB** active and click "Run" again, Divooka will re-evaluate `LoadImage` and `Resize` again (unless you cache them), and then run `ExtractFeaturesB` and `ClassifierB`.

- **Tip:** If resizing the image is expensive and you do not need to recompute it, click "Cache" on **Resize**. Then, whichever classifier(s) you activate will reuse the same resized image buffer.

Scenario C: Depth-First Subtleties with Side Effects

Suppose you attach a **LogToFile** node immediately after "ExtractFeaturesA" - this node writes a small JSON file for debugging. Meanwhile, you also have **LogToFile** attached after "ExtractFeaturesB." When you activate both **ClassifierA** and **ClassifierB**, Divooka's depth-first ordering might cause "ExtractFeaturesA → LogToFile" to run entirely before it even begins "ExtractFeaturesB → LogToFile." If your downstream analysis script expects logs from B first, it may be surprised. The key takeaway is:

> **Do not rely on the internal order in which parallel branches execute.** If the relative timing of side effects matters, try to isolate them in a serial pipeline (or use a Procedural subroutine for guaranteed sequencing).

3.5 Summary

In Divooka's Dataflow Context, **execution order** is always inferred from data dependencies rather than explicit step numbers. By clicking a node's **Run** button, you request that it become Active - thereby triggering recompilation of all upstream inputs unless those nodes are cached. Divooka's runtime currently uses a depth-first strategy to sort and evaluate dependencies, but future versions may change. Toggling **Cache** on any node freezes its last output, prevents recomputation, and "prune" s away its dependencies during evaluation. Understanding how activation and caching interact allows you to build responsive, efficient dataflow graphs and avoid unexpected recomputation or hidden dependency pitfalls.

Chapter 4. Practical Applications

> A programming language without practical libraries is like a paintbrush without bristles - beautiful in theory, but useless in the real world.

Welcome to Chapter 4, where we step off the drawing board of visual programming and into the workshop of everyday tasks. Now that you understand the fundamentals of Divooka's graph-based syntax, it's time to see how those nodes translate into genuine, hands-on work. Whether you need to read and write files, wrangle rows and columns of data, tap into a database, or apply a slick image filter, Divooka's standard libraries let you accomplish each task without typing a single line of traditional code.

Think of this chapter as your utility belt: each section introduces a different tool from Divooka's toolkit. We start by showing you how to treat the file system like a friendly assistant - reading logs, writing reports, and moving files around with drag-and-drop ease. Next, you'll discover the `DivookaDataGrid`, which transforms tabular data (spreadsheets, CSVs, or query results) into a live object you can filter, group, and export at will. From there, we'll connect you to the wider world of databases - SQLite, PostgreSQL, and any ODBC source - so you can pull down data, run queries, and push results back, all within the same visual environment. Finally, for anyone who's ever wanted to automate a photo resize or add a cinematic vignette, our `OneShotProcessing` library offers dozens of image nodes - every flip, crop, and color tweak you could imagine, each exposed as a simple graph node.

By the end of this chapter, you'll see how Divooka isn't just a fun way to draw algorithms; it's a practical, full-featured platform for real-world workflows. Let's dive in - and begin with File I/O.

4.1 File I/O

Divooka's standard library surfaces the most common file-system operations as simple "nodes" you can drop into your graph. Under the **Operating System → File System** category, you'll find nodes that mirror C#'s `System.IO` methods. Below is a high-level summary of the key nodes and how you might use them. Wherever possible, think of each node as a black-box: you supply the inputs (e.g., a path, text, or byte array), and the node emits the output (e.g., a success flag, a string array, or nothing at all).

Figure 4.1: *(Placeholder for a screenshot of the Divooka canvas showing a few File System nodes connected)*

4.1.1 File System Nodes

Node	Inputs	Outputs	Behavior
Append All Lines	**Path** (`string`), **Lines** (`string[]`), *(optional)* **Encoding**	-	Appends each string in **Lines** to the file at `Path`. If the file doesn't exist, it will be created.
Read All Lines	**Path** (`string`), *(optional)* **Encoding**	**Lines** (`string[]`)	Reads every line of the file at **Path** into a string array.
Read All Text	**Path** (`string`), *(optional)* **Encoding**	**Contents** (`string`)	Reads the entire contents of a text file into a single string.
Write Text To File	**Path** (`string`), **Text** (`string`), *(optional)* **Encoding**	-	Creates (or overwrites) the file at **Path** and writes **Text** into it as UTF-8 (unless another encoding is specified).
Copy	**SourcePath** (`string`), **DestinationPath** (`string`), *(optional)* **Overwrite** (`bool`)	-	Copies an existing file from **SourcePath** to **DestinationPath**. `If` **Overwrite** = true and the destination exists, it will be replaced.
Delete File	**Path** (`string`)	-	Deletes the file at `Path`. If the file does not exist, the node throws an error (unless wrapped in a "Try/Catch" subgraph).
File Exists	**Path** (`string`)	**Exists** (`bool`)	Returns **true** if the file at **Path** exists; otherwise, `false`.
Get Files	**DirectoryPath** (`string`), *(optional)* **SearchPattern** (`string`), *(optional)* **SearchOption** (`enum`)	**Files** (`string[]`)	Returns an array of file paths matching **SearchPattern** under `DirectoryPath`.

Node	Inputs	Outputs	Behavior
Move File	**SourcePath** (`string`), **DestinationPath** (`string`), *(optional)* **Overwrite** (`bool`)	-	Moves (or renames) a file from **SourcePath** to **DestinationPath**. If **Overwrite** = true, any existing file at the destination is replaced.
Get File Md5	**Path** (`string`)	**Md5Hash** (`string`)	Reads the file at **Path** and returns its MD5 hash as a hexadecimal string.
Touch	**Path** (`string`), *(optional)* **Size** (`uint`)	-	Creates a new empty file at **Path**, or if it already exists, truncates it to a length of **Size** bytes (default: 0).
Change File Extension	**Path** (`string`), **NewExtension** (`string`)	**ChangedPath** (`string`)	Changes the extension of a given file path without modifying the actual file.

Example Usage (C# equivalent)

- **Append All Lines**

```
File.AppendAllLines("logs.txt", new[] { "Log entry 1", "Log entry 2" },
Encoding.UTF8);
```

- **Read All Lines**

```
string[] lines = File.ReadAllLines("data.csv", Encoding.UTF8);
```

- **Read All Text**

```
string content = File.ReadAllText("config.json");
```

- **Write Text To File**

```
File.WriteAllText("output.txt", "Hello, Divooka!", Encoding.UTF8);
```

- **Copy**

```
File.Copy("data.txt", "backup/data_backup.txt", overwrite: true);
```

- **Delete File**

```
File.Delete("temp.txt");
```

- **File Exists**

```
bool found = File.Exists("data.db");
```

- **Get Files**

```
string[] logs = Directory.GetFiles("logs", "*.log", SearchOption.TopDirectoryOnly);
```

- **Move File**

```
File.Move("oldname.txt", "newname.txt", overwrite: true);
```

- **Get File Md5**

```
using var md5 = MD5.Create();
using var stream = File.OpenRead("archive.zip");
string hash = BitConverter.ToString(md5.ComputeHash(stream)).Replace("-",
"").ToLowerInvariant();
```

- **Touch**

```
// Create an empty file or truncate existing to zero length
File.Create("newfile.txt").Dispose();
```

- **Change File Extension**

```
string newName = Path.ChangeExtension("report.csv", ".bak");
// newName == "report.bak"
```

4.1.2 Path & Directory Nodes

Node	Inputs	Outputs	Behavior
Combine	**Paths** (`string[]`)	**FullPath** (`string`)	Joins multiple path segments into one valid file path, inserting the correct directory separators (`\` on Windows, `/` on Linux).
Get File Name Without Extension	**Path** (`string`)	**FileNameNoExt** (`string`)	Strips off the directory and extension, returning just the base file name.
Is Path Fully Qualified	**Path** (`string`)	**IsAbsolute** (`bool`)	Returns `true` if `Path` is rooted to a drive (e.g., `"C:\"`) or UNC share (`"\\server\share"`).

Node	Inputs	Outputs	Behavior
Get Temp File Name	-	**TempFilePath** (`string`)	Creates an empty 0-byte file with a unique name in the system temporary folder and returns its path.
Directory Exists	**DirectoryPath** (`string`)	**Exists** (`bool`)	Returns `true` if the directory exists; otherwise, `false`.
Get Directories	**DirectoryPath** (`string`), *(optional)* **SearchPattern** (`string`)	**Directories** (`string[]`)	Returns all subdirectory paths in `DirectoryPath` that match `SearchPattern`.
Get Creation Time	**Path** (`string`)	**CreationTime** (`DateTime`)	Retrieves the creation timestamp for a file or directory.
Set Creation Time	**Path** (`string`), **NewCreationTime** (`DateTime`)	-	Sets the creation timestamp on a file or directory.
Get Last Write Time	**Path** (`string`)	**LastWriteTime** (`DateTime`)	Retrieves the "last modified" timestamp for a file or directory.
Set Last Write Time	**Path** (`string`), **NewLastWriteTime** (`DateTime`)	-	Modifies the "last modified" timestamp for a file or directory.

Example Usage

- **Combine**

```
string full = Path.Combine("C:\\Users", "Alice", "Documents", "notes.txt");
```

- **Get File Name Without Extension**

```
string name = Path.GetFileNameWithoutExtension("C:\\temp\\image.png"); // "image"
```

- **Is Path Fully Qualified**

```
bool absolute = Path.IsPathFullyQualified("../logs.txt"); // false
```

- **Get Temp File Name**

```
string temp = Path.GetTempFileName();
```

- **Directory Exists**

```
bool dirFound = Directory.Exists("C:\\Data");
```

- **Get Directories**

```
string[] subs = Directory.GetDirectories("C:\\Projects", "*Divooka*");
```

- **Get Creation Time**

```
DateTime created = File.GetCreationTime("report.docx");
```

- **Set Creation Time**

```
File.SetCreationTime("report.docx", new DateTime(2025, 1, 1));
```

- **Get Last Write Time**

```
DateTime modified = File.GetLastWriteTime("report.docx");
```

- **Set Last Write Time**

```
File.SetLastWriteTime("report.docx", DateTime.Now);
```

4.1.3 Console & Environment Nodes

Although less common in purely visual workflows, the **Console** and **Environment** categories provide quick ways to interact with your runtime environment. **Get Environment Variable** is very commonly used when you need to hide your secrets (like API keys).

Node	Inputs	Outputs	Behavior
Print	**Value** (`any type`)	-	Converts `Value` to its string form and writes it to Divooka's console window (or application's standard output).
Get Environment Variable	**VariableName** (`string`)	**VariableValue** (`string`)	Reads the named environment variable (e.g., `"PATH"` or `"USERPROFILE"`) and returns its value, or an empty string if it doesn't exist.
Get Current Directory	-	**DirectoryPath** (`string`)	Retrieves Divooka's working directory at runtime.
Set Current Directory	**DirectoryPath** (`string`)	-	Changes Divooka's working directory at runtime.

Example Usage

- **Print**

```

```

```
Console.WriteLine("Processing completed.");
```

- **Get Environment Variable**

```
string pathVar = Environment.GetEnvironmentVariable("PATH");
```

4.2 Tabular Data Processing

Divooka's **DivookaDataGrid** is a first-class data structure for in-memory tables (rows × columns). Think of it as a lightweight spreadsheet object: once you have a `DivookaDataGrid` (for example, from reading a CSV, or from a database query), you can manipulate/filter/transform it using built-in nodes without having to write SQL or loop through rows manually. Under the hood, most of these grid methods simply wrap C# code (via reflection) and, in some cases, spin up an in-memory SQLite instance to power quick SQL queries.

Figure 4.2: *(Placeholder for a screenshot showing a DivookaDataGrid with sample data and some transformation nodes attached)*

4.2.1 Getting Data In and Out

Node	Inputs	Outputs	Behavior
Rows (property)	*(none)*	**RowArray** (`IDictionary<string,object>[]`)	Returns each row as a dictionary mapping column names to values. Useful when you need to loop over rows in your graph (e.g., feed into a "For Each" node).
RowValues (property)	*(none)*	**TwoDArray** (`object[][]`)	Returns a 2D jagged array of raw cell values. Each inner `object[]` corresponds to one row, in the same column order as `DivookaDataGrid.Columns`.
ToCSVText	**WithColumnHeader** (`bool`, default=`true`)	**CsvString** (`string`)	Converts the entire grid into a properly escaped CSV string. If `WithColumnHeader` = true, the first line contains column names. Duplicate column names are automatically disambiguated.

Node	Inputs	Outputs	Behavior
ToPrologFacts	**TableNameOverride** (`string`, nullable)	**PrologString** (`string`)	Generates a set of Prolog facts from the grid. Each row becomes a Prolog predicate of the form `table_name(col1, col2, …)`. Useful for feeding data into a Prolog engine for logic programming demos.
ToSQLiteCreateTable	**TableName** (`string`, nullable)	**CreateTableSql** (`string`)	Produces a `CREATE TABLE IF NOT EXISTS [TableName] (…)` statement in SQLite dialect, declaring every column as `TEXT`. If there are no columns, returns an empty string.
ToSQLiteInsertCommands	**TableName** (`string`, nullable)	**InsertStatements** (`string`)	Builds a series of `INSERT INTO [TableName] (col1, col2, …) VALUES (…)` statements for each row, escaping values appropriately.

> **Tip:** For a one-step SQLite export, you can call the obsolete `ToSQLite(string? tableName)` method. For best practice, generate the `CREATE TABLE` and `INSERT` statements separately (it's more transparent).

Example Workflows

- **ToCSVText**
  1. Drop in a `DivookaDataGrid` node containing your table (e.g., columns: `ID, Name, Score`).
  2. Add a **ToCSVText** node and leave `WithColumnHeader = true`.
  3. The output `CsvString` can be fed into a **Write Text To File** node to produce a CSV file with headers.
- **ToPrologFacts**
  1. Given a grid with columns `Person, Age, City`, add a **ToPrologFacts** node with `TableNameOverride = "People"`.
  2. The output `PrologString` will look like:

```
People("Alice", 30, "Seattle").
People("Bob", 25, "Toronto").
…
```

- **ToSQLiteCreateTable**
  1. Given a grid with columns `Product, Quantity, Price`, add a **ToSQLiteCreateTable** node with `TableName = "Inventory"`.
  2. The output `CreateTableSql` will be:

```
CREATE TABLE IF NOT EXISTS Inventory (
  Product TEXT,
  Quantity TEXT,
  Price TEXT
);
```

- **ToSQLiteInsertCommands**

  1. Using the same grid (`Product, Quantity, Price`), add a **ToSQLiteInsertCommands** node with `TableName = "Inventory"`.
  2. If the grid has two rows (`"Widget", 10, 2.5` and `"Gadget", 5, 3.0`), the output `InsertStatements` will be:

```
INSERT INTO Inventory (Product, Quantity, Price) VALUES ('Widget', '10', '2.5');
INSERT INTO Inventory (Product, Quantity, Price) VALUES ('Gadget', '5', '3.0');
```

4.2.2 High-Level Queries

Instead of manually writing SQL, Divooka provides convenience nodes - **Extract**, **Filter**, **GroupBy**, and **GroupCut** - that let you accomplish most data-analysis tasks visually. Under the **DivookaDataGrid** → **EasyQuery** category, you'll find:

Node	Inputs	Outputs	Behavior
Extract	`DataGrid` (`DivookaDataGrid`), `Expression` (string, e.g. `"Name, Age * 2 as DoubleAge"`)	`ResultGrid` (`DivookaDataGrid`)	Performs `SELECT <Expression> FROM "<TableName>"`. You don't have to write the full `SELECT … FROM …` clause - just list the columns and expressions you want.
Filter	`DataGrid` (`DivookaDataGrid`), `Condition` (string, e.g. `"Age > 30, Country = 'Canada'"`)	`FilteredGrid` (`DivookaDataGrid`)	Under the hood it executes `WHERE <conditions>` (commas → `AND`). Quoted literals are preserved.
GroupBy (2-argument)	`DataGrid` (`DivookaDataGrid`), `Expression` (string, e.g. `"Department, SUM(Salary) as TotalPayroll"`), `By` (string, e.g. `"Department"`)	`GroupedGrid` (`DivookaDataGrid`)	Runs `SELECT <Expression> FROM "<TableName>" GROUP BY <By>`. If `Expression` is blank, it defaults to the same as `By`.

Node	Inputs	Outputs	Behavior
GroupBy (4-argument)	`DataGrid` (`DivookaDataGrid`), `Expression` (string, optional), `OptionalFilter` (string, optional), `By` (string), `AggregateCondition` (string, optional)	`GroupedGrid` (`DivookaDataGrid`)	Builds and executes: `SELECT <Expression or By> FROM "<TableName>" [WHERE <OptionalFilter>] GROUP BY <By> [HAVING <AggregateCondition>]` Useful if you want to filter rows first (e.g., only US sales) and then group.
GroupCut	`DataGrid` (`DivookaDataGrid`), `Attributes` (string, e.g. `"Region"`), `Dimension` (string, e.g. `"Age"`), `Slices` (string, e.g. `"5"` or `"0,10,20,30"`), `Function` (enum, e.g. `Count`)	`BinnedGrid` (`DivookaDataGrid`)	If you specify `Slices = "5"`, Divooka finds the min/max of `Age`, divides it into 5 equal bins, and counts the number of rows in each bin per `Region`. The output is a pivoted grid where each bin becomes its own column (e.g., `Age (0–20)`, `Age (20–40)`, etc.).

Example Workflows

- **Extract**
  1. Drop in a `DivookaDataGrid` node containing employee data (columns: `Name, Department, Salary`).
  2. Add an **Extract** node, set `Expression = "Name, Salary * 1.1 as AdjustedSalary"`.
  3. The output grid has only two columns: `Name` and `AdjustedSalary`.
- **Filter**
  1. Starting with a grid of sales records (columns: `Region, SalesAmount, Date`).
  2. Add a **Filter** node, set `Condition = "SalesAmount > 1000, Region = 'West'"`.
  3. The output grid contains only rows where `SalesAmount > 1000` *and* `Region == "West"`.
- **GroupBy (2-argument)**
  1. Have a grid of employee salaries: columns `Department, Salary`.
  2. Drop a **GroupBy** node with `Expression = "Department, SUM(Salary) as TotalPayroll"`, `By = "Department"`.
  3. Output grid contains two columns: `Department` and `TotalPayroll` (one row per department).
- **GroupBy (4-argument)**
  1. Grid: `Region, SalesAmount, Month`.
  2. Add a **GroupBy** node:
     - `Expression = "Region, AVG(SalesAmount) as AvgSales"`
     - `OptionalFilter = "Month = 6"`
     - `By = "Region"`
     - `AggregateCondition = "AVG(SalesAmount) > 5000"`

3. Result: For each `Region` in June (`Month = 6`), compute its average; only include those with `AvgSales > 5000`.

- **GroupCut**

1. Grid: `Country, Age, SalesCount`.
2. Add a **GroupCut** node:
    - `Attributes = "Country"`
    - `Dimension = "Age"`
    - `Slices = "4"`
    - `Function = Count`
3. Result: For each country, four columns appear:
    - `Age (min–quartile1)`
    - `Age (quartile1–quartile2)`
    - `Age (quartile2–quartile3)`
    - `Age (quartile3–max)` Each cell shows how many rows fell into that age range in that country.

4.2.3 Mid-Level/Raw SQL

If you already know SQL or need a custom join across multiple grids, Divooka lets you run raw SQLite directly.

Node	Inputs	Outputs	Description
PerformQuery	**DataGrid** (`DivookaDataGrid`), **Query** (`string`, must start with `SELECT` and reference `"<TableName>"`), *(optional)* **TableNameOverride** (`string`)	**ResultGrid** (`DivookaDataGrid`)	Runs a raw SQLite query on a single grid and returns the result as a new `DivookaDataGrid`.
SQL	**Overload 1 (Simplest):** • **Tables** (array of `DivookaDataGrid`) • **Query** (`string`) **Overload 2 (Explicit Names):** • **Tables** (array of `DivookaDataGrid`) • **TableNames** (array of `string`s) • **Query** (`string`)	**ResultGrid** (`DivookaDataGrid`)	Like **PerformQuery**, but imports multiple grids simultaneously so you can join or union across them.

PerformQuery

Behind the scene, **Perform Query** does the following:

1. Creates an in-memory SQLite database.
2. Imports your `DataGrid` as a table named `"<TableNameOverride>"` (or its built-in `TableName` if no override is provided).
3. Executes `CREATE VIEW "<TempResult>" AS <Query>`.
4. Reads back the view into a new `DivookaDataGrid` and emits it on `ResultGrid`.

Example Workflow:

1. Suppose you have a `DivookaDataGrid` node named `"SalesGrid"` with columns: `Product, Quantity, Price`.
2. Drop in a **PerformQuery** node and set:
    - `Query = "SELECT Product, Quantity * Price AS Revenue FROM \"SalesGrid\" WHERE Quantity > 10"`

- `TableNameOverride = "SalesGrid"` (leaving it blank defaults to the grid's own table name).

3. The output `ResultGrid` will be a new grid containing only the `Product` and `Revenue` columns for rows where `Quantity > 10`.

SQL (multiple overloads)

The SQL node provides direct SQL execution capabilities against the provided data grids.

- **Inputs (Overload 1 – Simplest):**
  - `Tables` (array of `DivookaDataGrid`)
  - `Query` (string)
    Divooka will assign table names `"Table1"`, `"Table2"`, ... in the order provided.
- **Inputs (Overload 2 – Explicit Names):**
  - `Tables` (array of `DivookaDataGrid`)
  - `TableNames` (array of strings)
  - `Query` (string)
    You can reference each grid in the SQL by the corresponding name in `TableNames`.

Behind the scene, **SQL** node does the following:

1. Creates an in-memory SQLite database.
2. Imports each `DivookaDataGrid` into its own table (named either `"Table1"`, `"Table2"`, ... or the explicit names in `TableNames`).
3. Executes the provided `SELECT` query, which can reference any of the imported tables.
4. Reads back the query result into a new `DivookaDataGrid` and emits it on `ResultGrid`.

Example Workflow (Two Grids):

1. Grid A (`Employees`): columns `ID, Name, DeptID`.
2. Grid B (`Departments`): columns `ID, DeptName`.
3. Add an **SQL** node with:
   - `Tables = [EmployeesGrid, DepartmentsGrid]`
   - `TableNames = ["Employees", "Departments"]`
   - `Query =`

```
SELECT e.Name, d.DeptName
FROM "Employees" e
JOIN "Departments" d ON e.DeptID = d.ID;
```

4. The output `ResultGrid` will be a new grid containing two columns: `Name` and `DeptName`, representing the join between employees and departments.

4.3 Database Access

Divooka provides built-in support for three types of databases - SQLite, PostgreSQL, and any ODBC-compliant source (for example, SQL Server, MySQL, Oracle, or even Excel via ODBC). Behind the scenes, each provider uses a different driver and expects its own connection-string format, but the visible nodes in Divooka look nearly identical. Once you have created a Connection object, you can send queries to the database and receive the results as a `DivookaDataGrid`.

> **Figure 4.3:** *(Placeholder for a screenshot showing a Divooka Database Explorer panel with available providers)*

4.3.1 Common Database Workflow

First, you select which provider you want to use. If you simply need a lightweight, file-based database for prototyping or local storage, SQLite is the easiest choice. When you need a full client/server RDBMS, you would pick PostgreSQL. If your data already lives in some other SQL engine - or even in an Excel spreadsheet accessed via ODBC - then the generic ODBC provider is the way to go.

Once the provider is chosen, the next step is to configure the connection. Every provider requires a connection string. For SQLite, that might look like:

(Show screenshot of connection to SQLite database with connection string, e.g. `Data Source=C:\path\to\mydb.sqlite;Mode=ReadWriteCreate;`)

In the PostgreSQL case, you would supply something along the lines of:

(Show screenshot of connection to PostgreSQL database with connection string, `Host=localhost;Port=5432;Username=myuser;Password=secret;Database=mydb;`)

And for an ODBC connection - say, to MySQL via its ODBC driver - you could write:

(Show screenshot of connection to ODBC service with DSN name, `Driver={MySQL ODBC 8.0 Driver};Server=localhost;Database=mydb;User=myuser;Password=secret;Option=3;`)

Most of the APIs for dealing with database are stateless - there is no explicit connection object. You just configure the database connection, then perform actions on it.

When you want to run a query, drop in the **Execute Query** node. You give it the `DbConnection`, the SQL text you want to run, and (if needed) a dictionary of named parameters. Under the hood, Divooka calls `DbCommand.ExecuteReader()`, retrieves every row and converts that into a `DivookaDataGrid`, and returns it to you.

At that point, you have a `DivookaDataGrid` full of rows and columns. If you want to filter, transform, or otherwise manipulate the tabular data, simply use any of the Tabular Data Processing nodes - Filter, Extract, EasyQuery, and so on - directly on that grid. If your goal is to export the data instead, you can call `ToCSVText()` on the grid, or loop through its `Rows` collection to write out exactly what you need.

After each operation is executed, Divooka will automatically dispose of the underlying database connection and free up any remaining resources.

4.3.2 Example: Querying a SQLite Database

Suppose you have a local SQLite file at `C:\Data\sales.sqlite` with a table called `Orders`. You want to retrieve all orders for the year 2024 and export them to a CSV.

1. **Create SQLite Connection**
    - Drag in a **Create SQLite Connection** node.
    - Set `ConnectionString = "Data Source=C:\\Data\\sales.sqlite;Mode=ReadOnly;"`.
    - Output: `DbConnection` node (named e.g. `sqliteConn`).
2. **Execute Query**
    - Add an **Execute Query** node.
    - Connect `Connection = sqliteConn`.
    - Set `SqlText = "SELECT OrderID, CustomerName, OrderDate, TotalAmount FROM Orders WHERE strftime('%Y', OrderDate) = '2024';"`.
    - Output: `DivookaDataGrid` node (named `orders2024`).
3. **Convert to CSV**
    - Drop a **ToCSVText** node.

- Connect `DataGrid = orders2024`.
- Leave `WithColumnHeader = true`.
- Output: `CsvString`.

4. **Write CSV to File**
   - Attach a **Write Text To File** node.
   - Connect `Path = "C:\\Exports\\orders_2024.csv"`.
   - Connect `Text = CsvString`.
5. **Close Connection**
   - Attach a **Close Connection** node, input `Connection = sqliteConn`.

At the end of this little subgraph, you have a file `orders_2024.csv` containing all your 2024 orders.

> **Tip:** If you need to insert or update data, you can also use **Execute Non-Query** nodes (e.g., `INSERT`, `UPDATE`, `DELETE` statements). The output of those nodes is typically the integer number of rows affected.

4.4 Image Processing

Divooka includes a custom library called **OneShotProcessing** that exposes a wide variety of single-step image edits. Each function comes in two flavors:

1. **File-based Overload**
   - Takes `OriginalPath` (`string`) and `OutputPath` (`string`), plus any parameters.
   - Performs the operation on disk.
2. **PixelImage Overload**
   - Takes a `PixelImage` object (an in-memory image buffer) and returns a new `PixelImage`.
   - Allows chaining multiple operations without re-writing to disk each time (faster for pipelines).

Figure 4.4: *(Placeholder for a screenshot showing a chain of OneShotProcessing nodes: LoadImage → ResizeImage → AdjustBrightnessContrast → SaveImage)*

Below is a categorized summary of the most commonly used image nodes and typical use cases.

4.4.1 Basic Transformations

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
FlipImageHorizontally	**OriginalPath**, **OutputPath**	**PixelImage**	Creates a mirror image (left↔right).
FlipImageVertically	**OriginalPath**, **OutputPath**	**PixelImage**	Creates a mirror image (top↔bottom).
RotateImageClockwise90Degrees	**OriginalPath**, **OutputPath**	**PixelImage**	Rotates the image 90° clockwise. Useful for orientation.
RotateImageCounterclockwise90Degrees	**OriginalPath**, **OutputPath**	**PixelImage**	Rotates the image 90° counterclockwise.

4.4.2 Resizing

Function (Variant)	File-based Inputs	PixelImage Inputs	Behavior/Description

Function (Variant)	File-based Inputs	PixelImage Inputs	Behavior/Description
ResizeImage (Proportional)	`OriginalPath`, `OutputPath`, `NewWidth` (`uint`)	`PixelImage`, `NewWidth` (`uint`)	Calculates `NewHeight` to preserve aspect ratio, then resizes.
ResizeImage (Fixed Dimensions)	`OriginalPath`, `OutputPath`, `NewWidth` (`uint`), `NewHeight` (`uint`), *(optional)* `FillColor` (`Color`)	N/A (use File-based overload only)	1. Rescales proportionally to `NewWidth`. 2. If intermediate height > `NewHeight`: center-crop. 3. If intermediate height < `NewHeight`: expand canvas to `NewHeight`, filling extra space with `FillColor` (default = white).

Example Workflow (Fixed Dimensions):

1. **Load Image** (via a hypothetical **LoadImage** node → outputs `PixelImage img`).
2. **ResizeImage** node:
   - `(PixelImage) Original = img`
   - `NewWidth = 800`
   - `NewHeight = 600`
   - `FillColor = Color.Black` → Outputs `PixelImage resizedImg`.
3. **SaveImage** node:
   - `(PixelImage) Input = resizedImg`
   - `OutputPath = "C:\\Images\\resized_800x600.png"`.

4.4.3 Cropping

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
CropImage	`OriginalPath`, `OutputPath`, `OffsetX` (`uint`), `OffsetY` (`uint`), `CropWidth` (`uint`), `CropHeight` (`uint`), `FillColor` (`Color`, default = white)	`PixelImage`, `OffsetX` (`uint`), `OffsetY` (`uint`), `CropWidth` (`uint`), `CropHeight` (`uint`), `FillColor` (`Color`, default = white)	1. Creates a blank canvas of size `(CropWidth × CropHeight)` filled with `FillColor`. 2. Renders the original image at position `(-OffsetX, -OffsetY)`, so that pixels outside the source become the background color.

Example Workflow (Crop a 200×200 square from (50, 50)):

```
CropImage(
  Original = "photo.jpg",
  Output = "photo_cropped.jpg",
  OffsetX = 50,
  OffsetY = 50,
  CropWidth = 200,
  CropHeight = 200,
  FillColor = Color.White
)
```

4.4.4 Brightness, Contrast, and Color Adjustments

Function	File-based Inputs	PixelImage Inputs	Behavior/Description

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
AdjustBrightnessContrast	**OriginalPath**, **OutputPath**, **BrightnessPercent** (`double`), **ContrastPercent** (`double`)	**PixelImage**, **BrightnessPercent** (`double`), **ContrastPercent** (`double`)	Adjusts brightness and contrast. `BrightnessPercent`: 0 = no change, negative = darker, positive = brighter. `ContrastPercent`: 0 = no change, negative = less contrast, positive = more contrast.
AdjustSaturation	**OriginalPath**, **OutputPath**, **SaturationPercent** (`double`)	**PixelImage**, **SaturationPercent** (`double`)	Adjusts saturation. `SaturationPercent`: 100 = original, <100 = desaturated, >100 = oversaturated.
AdjustHSL	**OriginalPath**, **OutputPath**, **HueDegrees** (`double`), **SaturationPercent** (`double`), **LuminancePercent** (`double`)	**PixelImage**, **HueDegrees** (`double`), **SaturationPercent** (`double`), **LuminancePercent** (`double`)	Adjusts hue, saturation, and luminance. `HueDegrees`: 0–360; `SaturationPercent` and `LuminancePercent` as percentages.
GammaCorrection	**OriginalPath**, **OutputPath**, **Gamma** (`double`)	**PixelImage**, **Gamma** (`double`)	Applies gamma correction. `Gamma`: >1 darkens, <1 brightens.
AdjustLevels	**OriginalPath**, **OutputPath**, **BlackPoint** (`byte`), **WhitePoint** (`byte`), **Gamma** (`double`)	**PixelImage**, **BlackPoint** (`byte`), **WhitePoint** (`byte`), **Gamma** (`double`)	Adjusts levels by remapping brightness. `BlackPoint`: 0–255; `WhitePoint`: 0–255; `Gamma`: controls midtones.

Example Workflow (Increase Contrast):

```
AdjustBrightnessContrast(
  Original = "portrait.png",
  Output = "portrait_highcontrast.png",
  BrightnessPercent = 0,
  ContrastPercent = 30
)
```

4.4.5 Blurring, Sharpening, and Noise

Function	File-based Inputs	PixelImage Inputs	Behavior/Description

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
GaussianBlur	**OriginalPath**, **OutputPath**, **Radius** (`double`), **Sigma** (`double`)	**PixelImage**, **Radius** (`double`), **Sigma** (`double`)	Standard soft blur. `Radius` controls the area; larger = more blur.
MotionBlur	**OriginalPath**, **OutputPath**, **Radius** (`double`), **Sigma** (`double`), **Angle** (`double`)	**PixelImage**, **Radius** (`double`), **Sigma** (`double`), **Angle** (`double`)	Simulates directional blur. `Angle`: 0° = horizontal, 90° = vertical.
Sharpen	**OriginalPath**, **OutputPath**, **Radius** (`double`), **Sigma** (`double`)	**PixelImage**, **Radius** (`double`), **Sigma** (`double`)	Sharpens by emphasizing edges. `Radius` and `Sigma` control intensity and spread.
UnsharpMask	**OriginalPath**, **OutputPath**, **Radius** (`double`), **Sigma** (`double`), **Amount** (`double`), **Threshold** (`double`)	**PixelImage**, **Radius** (`double`), **Sigma** (`double`), **Amount** (`double`), **Threshold** (`double`)	More customizable sharpening than **Sharpen**: **Amount** (`how much`), **Threshold** (`difference limit`).
ReduceNoise	**OriginalPath**, **OutputPath**, **Order** (`uint`)	**PixelImage**, **Order** (`uint`)	Applies noise reduction. `Order`: higher = stronger noise reduction.

4.4.6 Artistic & Stylized Effects

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
SepiaTone	`OriginalPath`, `OutputPath`, `Threshold` (`double`, default = 80%)	`PixelImage`, `Threshold` (`double`, default = 80%)	Applies a warm, brownish sepia effect.
HighContrastBAndW	`OriginalPath`, `OutputPath`, `ContrastBoost` (`double`, default = 30%)	`PixelImage`, `ContrastBoost` (`double`, default = 30%)	Converts to grayscale and then boosts contrast for a stark black-and-white look.
BleachBypass	`OriginalPath`, `OutputPath`, `DesatPercentage` (`double`, default = 50%), `ContrastBoost` (`double`, default = 10%)	`PixelImage`, `DesatPercentage` (`double`, default = 50%), `ContrastBoost` (`double`, default = 10%)	Partially desaturates (bleach) and then increases contrast, giving a high-contrast cinematic look.

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
ApplyVignette	`OriginalPath`, `OutputPath`, `Radius` (`double`), `Sigma` (`double`), `XOffset` (`int`), `YOffset` (`int`)	`PixelImage`, `Radius` (`double`), `Sigma` (`double`), `XOffset` (`int`), `YOffset` (`int`)	Darkens edges toward black in an elliptical shape around the center.
ApplyEmboss	`OriginalPath`, `OutputPath`, `Radius` (`double`), `Sigma` (`double`)	`PixelImage`, `Radius` (`double`), `Sigma` (`double`)	Adds an embossed relief effect.
ApplyOilPaintEffect	`OriginalPath`, `OutputPath`, `Radius` (`int`), `Sigma` (`double`)	`PixelImage`, `Radius` (`int`), `Sigma` (`double`)	Simulates an oil-painting look with brush-stroke texture.

4.4.7 Color Filters & LUT

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
RecolorImage	`OriginalPath`, `OutputPath`, `OverlayColor` (`Color`), `OpacityPercent` (`double`, 0–100)	`PixelImage`, `OverlayColor` (`Color`), `OpacityPercent` (`double`, 0–100)	Paints a translucent layer of `OverlayColor` over the image.
ApplyLUT	`OriginalPath`, `LutImagePath` (`string`), `OutputPath`	`PixelImage`, `LutImagePath` (`string`), `OutputPath`	Takes a 256×1 or 256×256 CLUT (color lookup table) image and remaps source colors accordingly (film-style grading).
AdjustSelectiveSaturation	`OriginalPath`, `OutputPath`, `TargetColor` (`Color`), `Tolerance` (`percentage`), `SaturationPercent` (`double`)	`PixelImage`, `TargetColor` (`Color`), `Tolerance` (`percentage`), `SaturationPercent` (`double`)	Only changes saturation of pixels within `Tolerance` of `TargetColor`; other colors untouched.

4.4.8 Utilities & Compositing

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
ConvertToGrayscale	`OriginalPath`, `OutputPath`	`PixelImage`	Converts to black-and-white (no grayscale ramp).
InvertColors	`OriginalPath`, `OutputPath`	`PixelImage`	Inverts every pixel (negative image).

Function	File-based Inputs	PixelImage Inputs	Behavior/Description
CompositeImages	`BaseImagePath`, `OverlayImagePath`, `OutputImagePath`, `Offset` (`Point: X, Y`), `CompositeOperator` (enum: Over, Multiply, Screen, etc.)	`PixelImage`-based overload not supported (use File-based version)	Uses the specified composite operator (e.g., Over) to blend `Overlay` onto `Base` at offset `(X, Y)`.
AddText	`OriginalPath`, `Text` (`string`), `Location` (`Point: X, Y`), `FontFamily` (`string`), `FontSize` (`int`), `Color` (`Color`), `OutputPath`	`PixelImage`, `Text` (`string`), `Location` (`Point: X, Y`), `FontFamily` (`string`), `FontSize` (`int`), `Color` (`Color`)	Renders `Text` at the given `Location`. File-based version also writes to `OutputPath`.
AddFilmGrain	`OriginalPath`, `NoiseType` (enum; e.g. Gaussian), `OpacityPercent` (`double`, default = 20%)	`PixelImage`, `NoiseType` (enum; e.g. Gaussian), `OpacityPercent` (`double`, default = 20%)	Overlays a noise layer at the given opacity to simulate film grain.
Pixelate	`OriginalPath`, `OutputPath`, `Size` (`int`, default = 4)	`PixelImage`, `Size` (`int`, default = 4)	1. Downsamples by factor `Size` (box filter). 2. Upscales back with nearest-neighbor (point filter), yielding a blocky, pixel-art look.
PixelateCircle	`OriginalPath`, `OutputPath`, `Size` (`int`, default = 4)	`PixelImage`, `Size` (`int`, default = 4)	Similar to **Pixelate**, but draws circles instead of square blocks, creating a dotted mosaic style.

4.5 Summary

In this chapter, we have introduced some core standard-library nodes that Divooka makes available out of the box. By borrowing heavily from the existing .NET system libraries and ecosystem, Divooka exposes:

1. **File I/O**: Everything from checking file existence (`File Exists`) to reading/writing lines or bytes (`Read All Lines`, `Write All Text`), directory enumeration, file renaming/moving, and simple MD5 checksums.
2. **Tabular Data Processing (`DivookaDataGrid`)**: An in-memory grid that can be filtered, projected, grouped, and exported without hand-writing loops. High-level nodes like **Extract** and **Filter** let you build queries visually, while raw **PerformQuery** or **SQL** nodes drop down into SQLite when you need full control.
3. **Database Access**: Transparent support for SQLite, PostgreSQL, and ODBC, so that any data you pull from a database can immediately become a `DivookaDataGrid`. You can then use the same grid-processing nodes you already know (Filter, Extract, etc.) to analyze relational data.
4. **Image Processing (OneShotProcessing)**: A rich set of single-step operations - flips, rotations, resizing, cropping, color adjustments, blurs, artistic filters, and compositing - all accessible as simple Divooka nodes. Use the file-based overloads for quick disk operations, or work with `PixelImage` overloads in memory for chaining multiple effects in one graph.

Whether you need to process text files, manipulate CSV-style data in memory, query or update a database table, or apply a cinematic filter to a photo, Divooka's standard package has a node for it. In the next chapter, we'll dive into how you can build reusable subgraphs (custom nodes) and package your own libraries for divisibility and sharing.

Chapter 5. Intermediate Topics

> At first, I thought visual programming was just a toy. Then I saw a graph generate insights, compile the report, and send the email - before I'd even finished my coffee.

As your graphs grow in complexity and ambition, you'll find yourself needing more than just basic data transformations. Whether it's executing a block of logic in precise order, automating tasks from the command line, or extending Divooka's capabilities with native code, this chapter introduces the tools that let your graphs operate at the next level. From procedural thinking to CLI automation, we now enter the world of power users.

5.1 Procedural Context

In Divooka, a **procedural context** supplements the usual node-based dataflow paradigm by allowing users to embed sequential, function-like constructs directly within a graph. Instead of treating every computation as a pure dataflow transformation, procedural contexts enable you to define "in-place" logic - such as event handlers, callbacks, or multi-step algorithms - using familiar programming constructs. Under the hood, the graph runtime recognizes these procedural sections as discrete execution blocks: when the graph is evaluated, procedural nodes within the same context run in sequence, while still interacting seamlessly with the overall dataflow.

One of the key motivations for adding procedural contexts is to support patterns that require **variable capture** and **delegates** without forcing users to create bespoke nodes for every possible callback type. For example, imagine you have a graph-native "map" node that applies a function to each element of a list. In pure dataflow, you'd need a separate node type for each unary function; with a procedural context, you can simply capture a lambda or delegate, attach it to the map operator, and let the graph runtime invoke it at each iteration. This approach is inspired by how functional languages like Haskell treat functions as first-class citizens: you can partially apply a function (e.g., `add 5`) and pass it around, then call it later with the remaining arguments. Divooka's procedural contexts emulate this by allowing you to bind variables "in place," so that a captured delegate lives alongside its environment and executes as part of the graph's run.

Procedural contexts also streamline **custom event handling** within a graph. Instead of wiring together dozens of specialized event nodes, you can insert a procedural block that listens for a specific trigger, evaluates a condition, performs side-effects, and then returns control to the main graph flow. Internally, Divooka treats these blocks as mini-subgraphs that the scheduler invokes in the correct order. This design makes it easier to implement patterns like "retry on failure," "debounce user input," or "aggregate results over time" without sprawling the node layout.

By combining dataflow semantics (for parallel or reactive processing) with procedural contexts (for sequential logic), Divooka gives developers the flexibility to choose the right abstraction for each part of their application. If you need pure transformation pipelines - such as bulk data processing - you lean on dataflow. If you need fine-grained control - such as handling asynchronous callbacks or stateful loops - you use a procedural context. The result is a unified "language of the graph" where both paradigms coexist, letting you express complex behaviors without jumping out into a separate scripting layer.

5.2 CLI Automation (Stewer)

Divooka's command-line launcher, **Stewer (stew)**, provides a lightweight way to invoke graphs from scripts, CI/CD pipelines, or operating-system schedulers. Essentially, Stew acts as a thin wrapper that loads a `.dvk` file (or legacy `.Parcel` format) and executes it with any specified inputs, then returns the results or error codes to the shell. This makes it straightforward to automate repetitive tasks - such as nightly data imports, scheduled reports, or batch AI inference - without having to open the Divooka IDE each time.

The most basic invocation is:

```
stew <Graph Path> [<Graph Inputs>]
```

Here, `<Graph Path>` is the file path to your graph definition (e.g., `Graphs/MyDataImport.dvk`), and `<Graph Inputs>` are optional positional parameters that feed into designated "input" nodes within the graph. For example, if your graph expects a single input (say, a CSV file path), you could call:

```
stew Graphs/ImportSalesData.dvk "C:\Data\sales_2025_06_01.csv"
```

to kick off the import process from the command line.

If you forget the syntax or want more details, simply run `stew --help`:

```
stew --help
```

This prints the generic usage information and explains the global options:

- `stew <Graph Path> [<Graph Inputs>]`: Launches the specified graph with optional inputs.
- `stew --help`: Displays the high-level Stew usage.
- `stew --help <Graph Path>`: Shows help specifically for that graph, including a list of input nodes and expected argument types.

For instance, if you need to know exactly which inputs "ImportSalesData.dvk" expects, you would run:

```
stew --help Graphs/ImportSalesData.dvk
```

and Stew will list each named input node, its data type (e.g., string, integer, file path), and any default value settings. This makes it easy to script around complex graphs, since you can dynamically discover their signatures without inspecting the `.dvk` file manually.

Because Stew returns standard exit codes (0 on success, non-zero on failure), you can embed these calls into batch files, PowerShell scripts, or even cron jobs (using Windows Task Scheduler). For example, a PowerShell snippet to run a graph nightly might look like this:

```
$graphPath = "C:\Divooka\Graphs\GenerateReports.dvk"
$outPath   = "C:\Reports\DailyReport_$(Get-Date -Format yyyy-MM-dd).pdf"

# Invoke the graph, passing the output file path as an input
stew $graphPath $outPath
if ($LASTEXITCODE -ne 0) {
    Write-Error "Report generation failed with code $LASTEXITCODE"
} else {
    Write-Output "Report generated: $outPath"
}
```

In CI/CD pipelines, you can similarly use Stew to run tests, validate data, or deploy artifacts. By placing `stew` commands in your build scripts, every commit to your main branch can trigger a sequence of Divooka graphs - such as "Run Unit Tests," "Build Documentation," and "Publish to Staging" - thereby integrating Divooka workflows into standard DevOps practices.

5.3 Plugin development (C#)

There are a few different ways you can extend Divooka. If you are using Divooka Compute, you can even extend nodes by using loose C# and Python scripts directly. For most distributions, however, the most ubiquitous way of providing additional toolbox is through a custom C# assembly.

At the moment a plugin can do the following:

1. Register new dynamic types and provision new toolboxes. This including registering specific tools and categories.
2. Register new base type for the procedural context.
3. Provide new menus for the (supported) GUI editor.
4. Register new preview types.

A plugin can provide new toolboxes or extend the graph editor itself.

5.3.1 Locating Plugins

All plugins all reside in the `Plugins` folder of a given distribution. Plugins (e.g. `MyPlugin`) must have all required files in its own plugin folder (e.g. `MyPlugin`) and have a same named entry assembly, so the path to a plugin looks like: `DivookaDistribution/Plugins/<PluginName>/<PluginName>.dll`.

A plugin may also have a different entry assembly, in which case it must provide `<PluginName>.package.yaml` file containing the following fields:

```
Identifier: <PluginName>
Display Name: <Human Friendly Plugin Name>
Entry Assembly: <Entry>.dll
```

The simplest form of plugin is just a plain C# class. Use a static class if you just wish to expose simple functions:

```
public static class MyPlugin
{
    public static int GetValue() => 15;
}
```

For more advanced and customized capabilities, one can implement `IDevookaEnginePlugin`. We will talk more about this in a latter section.

```
public interface IDevookaEnginePlugin
{
    #region Lifetime
    public void Initialize(IPluginServiceProvider provider);
    public void Shutdown(IPluginServiceProvider provider);
    #endregion
}
```

5.3.2 C# Tricks

When writing C# code for Divooka plugins, a few idiomatic patterns and XML annotations ensure that your methods and types appear cleanly in the graph editor. In Divooka, any method you intend to expose as a standalone node should be

declared as a `public static` function. Because static methods hold no instance state, Divooka automatically generates a node whose input pins correspond to the method's parameters and whose primary output pin reflects the method's return type. For example, consider a simple static method that computes a square:

```
/// <summary>
/// Returns the square of an integer.
/// </summary>
public static int Square(int x) => x * x;
```

When this code is loaded, Divooka presents a node named "Square" with one input (**x** of type `int`) and one output (an automatically named **Result** of type `int`), without requiring any extra wiring or boilerplate.

In addition to return values, *C#*'s `out` parameters are supported as additional output pins. If you declare a method like this:

```
/// <summary>
/// Divides an integer and returns quotient and remainder.
/// </summary>
public static int Divide(int dividend, int divisor, out int remainder)
{
    remainder = dividend % divisor;
    return dividend / divisor;
}
```

Divooka will create two outputs - one for the return value (quotient) and one for the `out` parameter (`remainder`). Crucially, `out` parameters do not appear as inputs on the node; instead, they automatically become extra output pins. This makes it straightforward to expose multi-value methods without cluttering the node's input side.

By contrast, instance methods are hidden by default unless you explicitly opt in. To expose an instance method in a pure-dataflow graph (so it behaves like a static function), add `<const>true</const>` in its XML documentation. For example:

```
public class MathHelpers
{
  /// <summary>
  /// Adds two numbers together.
  /// </summary>
  /// <const>true</const>
  public int Add(int a, int b) => a + b;
}
```

With `<const>true</const>`, Divooka treats `Add` as though it were static - displaying a node that takes two integers and returns their sum. Omitting `<const>` tells Divooka that the method likely depends on internal state or side effects, so it remains available only within procedural contexts, where an explicit object instance is maintained.

XML documentation serves as the source of truth for node metadata (tooltips, display names, input formats, and more). Divooka recognizes several tags and attributes:

- `<summary>` and `<remarks>` populate the node's tooltip or help panel, providing users with contextual guidance.
- `<param name="…" textFormat="…">` controls how Divooka renders or constrains that parameter - examples include `textFormat="text"` for plain text or `textFormat="Password"` to mask sensitive inputs.

- `<snap>URL</snap>` links to a screenshot that appears in the node's documentation pane, helping end-users understand usage at a glance.
- `<hidden>true</hidden>` prevents the method or type from appearing in the toolbox, even if it would otherwise be discoverable.
- `<const>true</const>` (on instance methods) forces Divooka to treat that method like a pure function in dataflow graphs.

For example, if you write:

```
/// <summary>
/// Executes a query and returns a DataGrid.
/// </summary>
/// <snap>https://cdn.example.com/snaps/QueryNode.png</snap>
public static DivookaDataGrid Query(string sql, ODBCConfiguration? config = null)
{
  // Implementation omitted…
}
```

Divooka will display "Executes a query and returns a DataGrid" as the tooltip, and it will show the referenced image when users inspect the node's documentation. If you add `<hidden>true</hidden>`, Divooka will hide that method from the user-facing toolbox entirely, even if other exposure conditions are met.

Divooka also supports delegate-based nodes: if a static or instance method includes a `Func<…>` or `Action<…>` parameter, that parameter becomes a lambda placeholder in the graph. For example:

```
/// <summary>
/// Applies a lambda to each element in an integer array.
/// </summary>
public static int[] Map(int[] items, Func<int, int> transformer)
{
    return items.Select(transformer).ToArray();
}
```

When dragged into a graph, "Map" shows an **items** input, a **transformer** subgraph placeholder, and an `int[]` output. Under the hood, Divooka compiles the user's subgraph into a delegate, applying it to every element at runtime.

Note that, as of Divooka 0.8.5, generic methods and classes are not supported. Even if you declare:

```
public static T Identity<T>(T value) => value;
```

Divooka ignores it. To expose functionality, you must use concrete types (for example, `int Identity(int)` or `string Identity(string)`). Future releases may add generic support, but for now, design your plugin API without type parameters.

Finally, Divooka does not recognize inheritance hierarchies or polymorphic dispatch when populating the toolbox. If you define a base class and a derived class that both implement `Compute(double)`, Divooka lists both separately; it will not combine them under a shared "Compute" node or perform runtime type-based dispatch. If you require polymorphic behavior, implement explicit wrapper methods or leverage procedural contexts to inspect types at runtime.

XML Tags and Attributes

Below is a summary of the XML tags and attributes most useful when annotating your C# code for Divooka. Use this as a quick reference when writing documentation comments.

Tag/Attribute	Purpose	Notes
`<summary>…</summary>`	Provides the node's main description.	Appears as tooltip text in the graph editor.
`<remarks>…</remarks>`	Offers additional explanatory text.	Shown in the detailed documentation pane, below the summary.
`<param name="…" …>…</param>`	Describes a parameter and (optionally) its input format.	Attributes: `textFormat="text"` (plain text), `textFormat="Password"` (masked input), etc.
`<snap>URL</snap>`	Embeds a screenshot or illustration URL.	Divooka displays this image in the node's documentation pane for visual guidance.
`<hidden>true</hidden>`	Hides the method or type from the toolbox.	Even if other criteria expose it, this tag forces Divooka to omit it from the UI.
`<const>true</const>`	Marks an instance method as a pure function in dataflow contexts.	Without this, instance methods are available only inside procedural contexts.
`<displayName>…</displayName>`	Overrides the automatically inferred display name for a node.	If omitted, Divooka uses the CLR-name; providing this tag lets you specify a user-friendly label.
`<returns name="…">…</returns>`	Names or describes the return type.	Rarely needed - Divooka infers return types automatically; use this only to declare custom return pin names.

Use these tags consistently to give users a polished, self-documenting experience when they drag your custom nodes into a graph.

5.3.3 The Plugin Framework[^3]

A `IDevookaEnginePlugin` can register functionality under five different areas: **menus**, **node appearance**, **previews**, **dynamic type registration**, and **execution runners**. It can query the host system which features are available by calling `GetFeatureSet()` on .

- **Menus**
  - Use `RegisterMenu`/`RegisterSubmenu` to add single-click actions.
  - Always supply a unique key (e.g. `"MyPlugin/Tools/DoSomething"`).
- **Node Appearance**
  - Use `RegisterTypeConnectionAppearance(typeof(MyType), BuiltinConnectorShape.SolidCircle, Color.Blue)` to give nodes for `MyType` a blue circular pin.
- **Preview Support**
  - If your node outputs a custom data type, implement `Func<object, object>` to convert it into an existing previewable object (string, Bitmap, array, etc.).
  - If you need a popup editor or something that Divooka can't preview natively, implement a custom preview handler.
- **Dynamic Types (Toolbox)**
  - Supply one or more toolbox definitions in `RegisterPackages`.
  - If you simply want to scan an entire assembly, use `AssemblyToolboxDefinition`. If you want tight control over individual node metadata, use `TypeToolboxDefinition`.

- **Execution**
  - If your plugin includes nodes that must run in a specialized context, register a `ProceduralContextExecutionRunner` mapping your graph's base type to a runner/factory.

Once the above pieces are in place, Divooka will automatically show your menus, render your node pins, let users drag your custom node types into graphs, and - when a user clicks "Run" or requests a preview - invoke your converters or custom preview code under the hood.

Plugins go through life cycle management in those events:

1. `Initialize()`
2. `Shutdown()`

One can use `RegisterMenu(... autoUnregister: true)` to automatically manage plugin deregistration if no custom resources need releasing.

```
public void RegisterMenu(string menu, string title, string? tooltip,
Action<IDevookaEngineHostEnvironment> clickCallback, bool autoUnregister);
```

Plugin Feature Flags

The host exposes a set of capabilities through the `PluginFeature` flags. Before attempting to use a particular subsystem, a plugin should call `GetFeatureSet()` and check for the relevant flag. The available features are:

- **Menus (1)**: The host can create new top-level menus and nested submenus.
- **DynamicPackages (2)**: The host can add new node-types (often called "packages") into Divooka's toolbox at runtime.
- **EditorQuery (4)**: The host allows querying and modifying the runtime state of graphs (e.g., reading node values, connecting/disconnecting pins).
- **ConnectorVisualCustomizations (8)**: The host permits customizing the appearance of node pin connections (shape + color).
- **CustomNodeSurface (16) [Obsolete]**: Planned but not yet implemented: full node surface customization (beyond built-in pin/connector rendering).
- **PreviewProcessor (32)**: The host supports registering routines that transform arbitrary data into a type that Divooka can preview natively (e.g., image, text, graph).
- **CustomPreviewControl (64) [Obsolete]**: Planned but not yet implemented: embedding a custom preview control (e.g., custom WPF or Godot control).
- **CustomPreviewHandler (128)**: The host supports a completely custom "popup" preview handler for any node type (plugin takes full control of how preview is rendered in a separate window).

A plugin should combine these flags with bitwise operations (e.g., `if ((features & PluginFeature.Menus) != 0) { … }`) to determine what to register or enable at runtime.

Menu Registration

Once the plugin has verified that `PluginFeature.Menus` is available, it can add items to Divooka's main menu bar and register nested submenus. Each menu entry invokes a callback when clicked.

```
// Add or remove a top-level menu item:
void RegisterMenu(
    string menu,                  // unique identifier (e.g. "MyPlugin/Tools")
    string title,                 // user-facing text
    string? tooltip,              // optional hover text
```

```
    Action<IDivookaEngineHostEnvironment> clickCallback,
    bool autoUnregister           // if true, host will remove it on plugin unload
);

void UnregisterMenu(string menu, string title);

// Add or remove a submenu under an existing menu:
void RegisterSubmenu(
    string menu,                  // same "menu" key as above
    string submenu,               // unique identifier for the submenu entry
    string title,
    string? tooltip,
    Action<IDivookaEngineHostEnvironment> clickCallback,
    bool autoUnregister
);

void UnregisterSubMenu(string menu, string submenu, string title);
```

Node Connector Appearance

If `ConnectorVisualCustomizations` is supported, a plugin can customize how pins (input/output ports) and connector lines look for any given data type.

```
void RegisterTypeConnectionAppearance(
    Type type,                    // the .NET type to style (e.g., typeof(MyCustomData))
    BuiltinConnectorShape shape,  // one of: HollowCircle, SolidSquare, etc.
    Color color                   // base color for the connector (e.g.,
System.Drawing.Color.Red)
);
```

- `BuiltinConnectorShape` A small enum of built-in shapes (HollowCircle, HollowRectangle, HollowTriangle, HollowArrow, HollowSquare, and their "Solid" counterparts).
- **Usage** Register this early (e.g. during plugin initialization). Divooka will then render all pins carrying that CLR type using your chosen shape + color.

Preview Processing

Divooka allows nodes to provide a small preview of their data (e.g., showing an image, a text snippet, or a graph). Two different extension points exist:

1. **Preview Processor** (requires the `PreviewProcessor` flag)

   ```
   void RegisterPreviewProcessor(
       Type type,                    // the data type your plugin wants to handle
       Func<object, object> processor // a function that converts "type" into a built-in
   previewable type (e.g., Bitmap, string, List<float>, etc.)
   );
   ```

   - **When to use**: If your plugin's node outputs a custom data type that Divooka cannot render natively, implement a processor that converts it into something the host already knows how to display.
   - **Example**: Converting a `MyImageData` instance into a `System.Drawing.Bitmap`.

2. **Custom Preview Handler** (requires the `CustomPreviewHandler` flag)

```
void RegisterCustomPreviewHandler(
    Type instanceType,                // the .NET type for which you supply full
control
    Action<ProcessorNode, object> previewHandler
);
```

- **When to use**: If you need a completely custom popup preview (e.g., an interactive chart or 3D view) that the host cannot handle.
- **Behavior**: Divooka will invoke your `previewHandler` whenever the user requests a preview; it passes you the node instance plus the raw data object. Your code is responsible for showing a window or control, rendering content, wiring up close events, etc. Note: this only works as a separate popup, *not* inline on the canvas.

Dynamic Type/Toolbox Registration

If `DynamicPackages` is enabled, the plugin can tell Divooka about new node types - both in terms of back-end functionality (what the node does) and front-end toolbox entries (where it appears in the UI). The `RegisterPackages` call unifies several collections of definitions:

```
void RegisterPackages(
    IEnumerable<ToolboxIndexer.AssemblyToolboxDefinition>? assemblyToolboxes,
    IEnumerable<ToolboxIndexer.FrontendToolboxDefinition>? frontendToolboxes,
    IEnumerable<ToolboxIndexer.TypeToolboxDefinition>? typedToolboxes,
    IEnumerable<ToolboxIndexer.SpecificToolboxEndPointDefinition>? specificEndpoints,
    IEnumerable<ToolboxIndexer.ProceduralContextBaseTypeDefinition>?
proceduralContextTypes,
    bool autoUnregister = true
);
```

- `AssemblyToolboxDefinition` Defines a whole assembly of nodes - Divooka can scan an assembly at runtime to find classes annotated as nodes, then automatically populate the toolbox.
- `FrontendToolboxDefinition` Describes how you want to group and label nodes in Divooka's UI (e.g., folder icons, categories).
- `TypeToolboxDefinition` Explicitly registers a single .NET type as a node - useful if you want fine-grained control over the name, icon, and where it appears.
- `SpecificToolboxEndPointDefinition` Allows you to place a node in more than one location or under a specialized context menu.
- `ProceduralContextBaseTypeDefinition` If your plugin provides a "procedural context" (a stateful environment for running a set of nodes in a custom way), you register the base type here so Divooka knows about your runtime execution model.

Once registered, Divooka's UI will show these new node types in the toolbox; users can drag and drop them onto the canvas just like built-in nodes.

Procedural Context Execution Runner

For plugins that introduce a custom procedural execution context, use:

```
void RegisterProceduralContextExecutionRunner(
    Type baseType,            // the abstract base for all of your procedural contexts
    Type instantiatorType     // a factory/runner class that knows how to execute that
```

```
context
);
```

- **Purpose** Tells Divooka: "Whenever a node graph rooted in a type derived from `baseType` needs execution, use this `instantiatorType` to run it."
- **Behavior** Divooka will instantiate your runner at runtime and hand it the node graph; your runner is responsible for traversing nodes, evaluating them in the correct order, and producing outputs.

5.3.4 Other Ways of Extending Divooka

So far we have seen ways to extend Divooka Engine (and derived software) through engine plugin. For the purpose of sharing and extending the capabilities of Divooka, there are other ways.

In this book we will not talk about more advanced forms of plugin development e.g. distributing as NuGet packages or publish to Divooka Central. Take a look at the section on Sharing for more details.

5.4 Summary

In this chapter, we explored two core aspects of intermediate Divooka use. First, **procedural contexts** bridge the gap between pure dataflow and sequential logic by allowing graph-native lambdas, event handlers, and captured variables to execute in order. This hybrid model empowers you to tackle problems that require fine-grained flow control - like custom retry loops or complex callback chains - while still benefiting from Divooka's visual composition. Second, the **Stewer CLI launcher** provides a powerful automation layer: by exposing graph execution to standard command-line workflows, you can integrate Divooka graphs into scripts, scheduled tasks, and CI/CD pipelines, enabling fully automated, repeatable processes. Together, these intermediate topics serve as a foundation for more advanced scenarios, such as writing custom C# plugins (covered in Section 5.3) or extending Divooka into production environments.

Chapter 6. Frameworks

In this section we are going to present two canonical frameworks. Among those, **Slide Present** is representative of the dataflow context while **Glaze!** is representative of procedural context.

6.1 Slide Present

Slide Present is a simple and opinionated framework for making presentations. It produces result similar to a typical PowerPoint presentation.

Fig. ?.? Slide Present Interface

The goal of this framework is to make it functional and allow creating presentations quickly and effectively.

As of version 0.2, the cosmetics aspects and configuration options are still a work-in-progress. Also notice that depending on the implementation you may see slightly different look. This is intentional, so you should generally not depend on it for exact looks; However as we move towards greater consolidation between implementations, we target a more uniform and predictable look.

6.1.1 Overview of Slide Present

When using Slide Present, the entire presentation is constructed as a single Divooka graph and it supports advanced features like embedding a Divooka graph directly in the presentation. Additional benefits include runtime constructed media contents (e.g. images).

Slide Present assumes a flat asset layout and uses explicit file references, this way when modifying the source file, one can immediately see the impact on the final presentation. Below screenshot shows a folder containing a presentation

that shows a few images and has audio and video.

Fig. ?.? Slide Present Assets Layout (in File Explorer)

If you are satisfied with a presentation, you can also export it into Markdown, PDF or PowerPoint formats - this requires corresponding extensions. However, notice only supported contents will be visible in those formats. E.g. Videos won't show up properly in text formats.

6.1.2 Presentation Construction

A presentation consists of the following components, each representing distinct objects for the presentation:

1. Presentation: The top level representation of the entire presentation.
2. Slide Section: A grouping of slides. Used as organization unit.
3. Slide: Slides contains contents and defines layout for each presentation screen.

In the Slide Present framework, slides have predefined layouts, which makes constructing "typical" slides easier. However, you have full control and can completely customize the look of a slide as well.

6.1.3 Preset Slide Layouts

Those are the few slide layouts that represent the most common scenarios:

1. Hero: Can be used to make titles, headers, or showing full screen images and videos.
2. One Body (optionally with header): Can be used to make typical single body content presentations.
3. Two Body (optionally with headers): Can be used to make slides containing two columns.
4. Multi-Body: Can be used to make slides containing multiple columns.

Below shows the construction of specific slides and along an example screenshot.

Fig. ?.? Hero - Title Slide Construction Screenshot

Fig. ?.? Hero - Title Slide

Fig. ?.? Hero - Fullscreen Media Construction Screenshot

Fig. ?.? Hero - Fullscreen Media

Fig. ?.? One Body - Slide Construction Screenshot

Fig. ?.? One Body

Fig. ?.? Two Body - Top Down Layout - Slide Construction Screenshot

Fig. ?.? Two Body - Top Down Layout

Fig. ?.? Multi Body - Slide Construction Screenshot

Fig. ?.? Multi Body

6.1.4 Sharing Objects Between Slides

Because slides are just containers of contents, the lifetime of objects can be shared throughout a presentation during runtime.

Below example illustrates sharing a graph canvas between two slides:

Fig ?.?. Use The Same Graph Instance as Content for Both Slides

This is useful when you want to retain the changes made on the first slide and continue the discussion on the second slide. AN illustrative scenario is like below:

Fig. ?.?. Showing Two Slides: The left slide talks about basic construction of the program; The second slide continue the discussion on how to extend/use the program.

6.1.5 Customize Slide

To completely customize a slide, you need to use components from (Simply App) and make use of adaptive layout containers and controls....

You can also use GLSL for background and image shading...

You may also make use of Glaze for embedded interactive media or background or the entire slide screen in supoprted front-end implemented Slide Presents....

6.1.6 The Constructive GUI Use

Slide Present is intended to serve as a foundational library and its features will be greatly expanded in the full release. We also expect a dedicated GUI designer in the future.

6.2 Glaze!

6.2.1 Overview

Glaze is our interactive media framework that works in a procedural context. If you've ever tinkered with Processing, you'll feel at home here - Glaze is directly inspired by and borrows heavily from Processing's design. That said, it's not a line-for-line drop-in: the API names don't always match, and we've extended Glaze with built-in support for simple GUI components so you can whip up a basic windowed application without needing a separate UI library.

Glaze runs on top of SFML (Simple and Fast Multimedia Library), which means your Glaze sketches will work cross-platform on Windows, macOS, and Linux. You can use it for:

- **Quick demonstrations of procedural shapes:** Draw circles, rectangles, lines, and more, all with just a few commands.
- **Basic 3D displays:** Set up a 3D camera, draw simple geometry (boxes, spheres, meshes), and rotate or move them in space.
- **Simple GUI applications:** Buttons, sliders, text inputs, and callback hooks are built right in - no need to import a separate widget toolkit.

Because Glaze sits closer to SFML than to a full-blown game engine, you won't find fancy scene graphs or multi-screen window managers out of the box. You *can* build multi-screen apps (e.g., switch between different "scenes" or states), but if you're aiming for a complex, multi-window dashboard, you might want to layer on a more dedicated UI framework. That said, for building small visualization tools, teaching demos, or lightweight creative experiments, Glaze is a perfect fit.

Below, we'll walk through all the pieces you need to know to get started with Glaze inside Divooka: window creation, drawing shapes, handling images, playing audio, 3D basics, and using GUI controls with event callbacks.

6.2.2 Window Creation

Every Glaze "sketch" begins by creating a window. You typically do this in a special `Start()` function, which Divooka invokes once at the start. Here's a minimal example:

```
// In Divooka's code editor, create a file named MySketch.dvk or similar.
Start() {
    // Create a window that's 800 × 600 pixels, with the title "My Glaze Window"
    createWindow(800, 600, "My Glaze Window");

    // Optionally set a target frame rate (default is 60 FPS)
    frameRate(60);

    // Set a default background color (RGB 240, 240, 240 = light gray)
    background(240);
}
```

Fig. ?.? Basic Glaze! Setup

Key points:

- `createWindow(width, height, title)` opens a window of the specified size and with the given title.
- `frameRate(fps)` is optional; if you don't call it, Glaze will try to update as fast as the hardware allows (usually 60 FPS).
- `background(r, g, b)` sets the fill for the entire window. You can call this anywhere (often in `Update()`) to clear the screen each frame.

Once the window is created, Divooka will repeatedly call your `Update()` function (see Section 6.2.3) until the user closes the window or you explicitly call `exit()`. If the user clicks the "X" in the window title bar (or presses Alt+F4, etc.), Glaze will detect that and automatically stop the `Update()` loop for you.

6.2.3 Basic Drawing

Drawing shapes in Glaze is done with simple drawing commands. Unlike other immediate mode drawing APIs (e.g. GDI+, Processing), the drawing commands usually takes arguments that specifies both the filling color and stroke colors directly, instead of separate `Fill()` and `Stroke()`functions. Here's a simple example sketch that animates a bouncing circle and a static rectangle:

```
float ballX, ballY;      // Current position of the ball
float ballVX, ballVY;    // Velocity components
float ballRadius = 30;

Start() {
    createWindow(800, 600, "Bouncy Ball Demo");
    frameRate(60);

    // Start the ball in the center, moving diagonally
    ballX = width / 2;
    ballY = height / 2;
    ballVX = 3.5;
    ballVY = 4.0;
}

Update() {
    // Clear the screen with a white background
    background(255);
```

```
    // Draw a semi-transparent blue rectangle (static)
    fill(0, 0, 255, 128);   // RGBA: blue with 50% opacity
    noStroke();
    rect(100, 200, 200, 150);  // x, y, width, height

    // Update ball position
    ballX += ballVX;
    ballY += ballVY;

    // Bounce off left/right edges
    if (ballX - ballRadius < 0 || ballX + ballRadius > width) {
        ballVX = -ballVX;
    }
    // Bounce off top/bottom edges
    if (ballY - ballRadius < 0 || ballY + ballRadius > height) {
        ballVY = -ballVY;
    }

    // Draw the ball (red outline, no fill)
    noFill();
    stroke(255, 0, 0);   // Red stroke
    strokeWeight(4);
    ellipse(ballX, ballY, ballRadius * 2, ballRadius * 2);
}
```

Explanation of key drawing functions:

- `background(r, g, b)` or `background(gray)` clears the window to a solid color. It's usually the first call in `Update()`.
- `fill(r, g, b, a?)` sets the interior color for shapes; if you omit the fourth argument, alpha is assumed to be fully opaque.
- `noFill()` disables filling, so subsequent shapes draw only their outlines.
- `stroke(r, g, b)` sets the outline color; `noStroke()` disables outlines altogether.
- `strokeWeight(w)` controls the thickness of shape outlines (in pixels).
- `rect(x, y, w, h)` draws a rectangle with its top-left corner at (x, y).
- `ellipse(x, y, w, h)` draws an ellipse centered at (x, y), with width w and height h.
- `line(x1, y1, x2, y2)` draws a straight line from (x1, y1) to (x2, y2).
- `triangle(x1, y1, x2, y2, x3, y3)` draws a triangle connecting three points.
- `pushMatrix()` / `popMatrix()` let you translate/rotate/scale a local drawing context without affecting the rest of the scene. For example:

  ```
  pushMatrix();
  translate(400, 300);
  rotate(radians(frameCount));
  rect(-50, -25, 100, 50);
  popMatrix();
  ```

This code snippet will draw a rotating rectangle around the screen's center. You call `popMatrix()` at the end to restore the previous coordinate system.

There are other utility functions - `backgroundGradient()`, `bezier()`, `arc()`, `quad()`, etc. - but the ones listed above cover most 2D drawing needs. If you're already comfortable with Processing, you'll recognize how quickly you can port sketches over to Glaze, with only minor tweaks to function names and color order (Glaze uses RGB ordering for all color functions).

6.2.4 Image and Video Media

Want to display a photograph, sprite, or even play a video file? Glaze has you covered with simple calls that mirror Processing's approach, but under the hood rely on SFML's image and video decoding.

Loading and Displaying Images

```
Image myPortrait;

Start() {
    createWindow(640, 480, "Image Demo");
    background(0);

    // Load an image from the project's "assets" folder; automatic error reporting
    myPortrait = loadImage("assets/portrait.png");
}

Update() {
    background(50);

    // Draw the image at position (50, 50). By default, images are drawn at 1× scale.
    drawImage(myPortrait, 50, 50);

    // Draw a smaller copy in the corner (scaled to 100×100)
    drawImage(myPortrait, width - 120, 20, 100, 100);
}
```

- `Image loadImage(String path)` attempts to load an image file (PNG, JPEG, etc.) and returns an `Image` object. Glaze will print an error message (in the console) if the file is missing or invalid, but it won't crash the whole sketch.
- `drawImage(Image img, x, y)` draws the image at its original size with its top-left corner at (x, y). There's also an overloaded version: `drawImage(img, x, y, w, h)`, which scales the image to width w and height h.

Playing Video Clips

Underneath, Glaze uses SFML's `VideoStream` or a similar wrapper. The API is straightforward:

```
Video myClip;

Start() {
    createWindow(800, 600, "Video Playback");
    myClip = loadVideo("assets/demo.mp4");  // MP4, AVI, etc.
    myClip.play();                          // Start playing immediately
    myClip.setLoop(true);                   // Loop at end
}
```

```
Update() {
    // Draw the current video frame at (0, 0)
    drawVideo(myClip, 0, 0, width, height);

    // You can check if the video has finished or is paused:
    if (myClip.isFinished()) {
        // e.g., show a "Replay" button or restart playback
    }
}
```

- `Video loadVideo(String path)` returns a `Video` object.
- `myClip.play()` begins playback. There's also `pause()`, `stop()`, and `seek(timeInSeconds)`.
- `setLoop(bool)` toggles looping.
- `drawVideo(Video vid, x, y, w, h)` renders the video frame to the screen. If you omit w/h, it draws at native resolution.

If you need to grab a pixel buffer from a video frame (for custom shader effects, for instance), you can use `vid.getCurrentFrame()` to get an `Image` object, then manipulate it directly. However, for most demos, `drawVideo()` is sufficient.

6.2.5 Audio Play

Sound in Glaze is also easy. You can load a short effect (WAV, OGG) or a longer music track and control playback, volume, and looping. Here's a snippet that demonstrates background music with a sound effect triggered by the user's mouse click:

```
Sound bgMusic;
Sound sfxLaser;

Start() {
    createWindow(640, 360, "Audio Demo");

    // Load background music (e.g., .ogg or .wav); set volume to 50%
    bgMusic = loadSound("assets/background.ogg");
    bgMusic.setLoop(true);
    bgMusic.setVolume(0.5);
    bgMusic.play();

    // Preload a short laser sound effect
    sfxLaser = loadSound("assets/laser.wav");
}

Update() {
    background(30);
    textSize(18);
    fill(255);
    text("Click anywhere to fire a laser!", 20, 30);
}

void mousePressed() {
    // Play the laser effect at full volume; multiple instances can overlap
    sfxLaser.play();
}
```

- `Sound loadSound(String path)` loads the audio file.
- `setLoop(bool)` toggles looping.
- `setVolume(float 0–1)` sets playback volume.
- `play()` starts playing immediately. You can also call `pause()` or `stop()`.
- If you call `loadSound()` multiple times on the same file, Glaze internally caches it so you're not reloading from disk each time.
- By default, calling `play()` on a `Sound` object launches a new "voice" each time. If you want to ensure that only one instance plays at a time (e.g., avoid stacking multiple background music tracks), call `stop()` before `play()`, or use `isPlaying()` to check if it's already running.

With these building blocks, you can layer music, effects, and narrative voice-overs as easily as clicking a few lines of code.

6.2.6 3D Contents

Processing fans will remember the P3D renderer; Glaze gives you a similar "3D mode" powered by SFML's OpenGL context. Before drawing any 3D geometry, you enter 3D mode by calling `begin3D()` (typically in `setup()`), then switch back to 2D if needed with `end3D()`. A simple rotating cube demo looks like this:

```
float angle = 0;

Start() {
    createWindow(800, 600, "3D Demo");
    frameRate(60);

    // Tell Glaze to use 3D (enable depth testing, etc.)
    begin3D();

    // Set up a perspective camera: fieldOfView(degrees), near, far
    perspective(60, 0.1, 1000);

    // Position the camera at (x, y, z), looking at (0, 0, 0), with upVector (0, 1, 0)
    camera(0, 0, 400,   0, 0, 0,   0, 1, 0);
}

Update() {
    background(50);

    // Enable depth testing each frame
    clearDepthBuffer();

    pushMatrix();
      // Move cube to the center
      translate(0, 0, 0);
      // Rotate over time
      rotateX(radians(angle));
      rotateY(radians(angle * 0.7));
      // Draw a wireframe cube of size 100
      noFill();
      stroke(0, 255, 0);
      strokeWeight(2);
      box(100);
    popMatrix();

    // Increment the angle
    angle += 1.2;
}
```

Explanation of key 3D functions:

- `begin3D()` switches Glaze into 3D rendering mode. It sets up a default depth buffer and enables OpenGL's depth testing.
- `end3D()` (if you ever want to go back to pure 2D) restores the orthographic 2D projection.
- `perspective(fovDegrees, nearClip, farClip)` defines a perspective projection.
- `camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)` sets position and orientation of the camera.
- `translate(x, y, z)`, `rotateX()`, `rotateY()`, and `rotateZ()` work just like their 2D counterparts but in three axes.
- `box(size)` draws a cube centered at the current coordinate origin with edge length = size.
- `sphere(radius, detail?)` draws a sphere of the given radius; you can optionally pass a detail level (number of subdivisions).
- `cone(bottomRadius, topRadius, height, detail?)`, `cylinder(radius, height, detail?)`, and `torus(innerRadius, outerRadius, detail?)` are also available for quick prototyping.
- If you need fully custom meshes, Glaze provides `beginShape(TRIANGLES)` / `vertex(x, y, z)` / `endShape()`, similar to Processing's `beginShape()`. Under the hood, it wraps SFML + OpenGL VBOs for you.

Because Glaze is still a relatively low-level approach, you won't find a scene graph or automatic frustum culling. If you plan to draw many objects or large models, you'll need to manage batching or switch to a dedicated 3D engine. But for teaching the basics of 3D transforms, creating fun rotating shapes, or mocking up quick visualizations, Glaze's 3D is more than enough.

6.2.7 GUI Controls and Events Callback

One area where Glaze extends beyond Processing is built-in support for basic GUI widgets. You can place buttons, sliders, checkboxes, and text fields directly in your sketch and attach event callbacks to them. This makes it trivial to create interactive tools (e.g., a color picker, parameter slider, or simple menu) without integrating a separate UI library.

Creating a Button

```
Button btnStart;

Start() {
    createWindow(500, 400, "GUI Demo");
    background(200);

    // Create a button at (20, 20), size 100×40, labeled "Start"
    btnStart = createButton(20, 20, 100, 40, "Start");

    // Assign a callback for when the button is clicked
    btnStart.onClick([]() {
        println("Start button was clicked!");
    });
}

Update() {
    // Clear background each frame (in case buttons redraw)
    background(220);

    // Draw the button. Glaze's internal UI system handles hover/pressed states
```

```
    btnStart.draw();
}
```

- `createButton(x, y, w, h, label)` returns a `Button` object.
- `btnStart.draw()` must be called every frame; Glaze internally handles hover highlighting and pressed animation.
- `btnStart.onClick(function)` registers a callback (lambda) that's invoked when the user presses and releases the button. You can also register `onHover()` or `onPress()` separately if you need more granular control.

Adding a Slider

```
Slider volumeSlider;

Start() {
    createWindow(400, 150, "Slider Demo");
    background(240);

    // Place a slider at (20, 60), width 200, height 20, ranging from 0 to 100, default at
50
    volumeSlider = createSlider(20, 60, 200, 20, 0, 100, 50);

    // Whenever the slider value changes, adjust the background music volume
    volumeSlider.onChange([](int newValue) {
        float vol = newValue / 100.0;  // convert to 0.0–1.0
        bgMusic.setVolume(vol);
        println("Volume set to " + newValue);
    });
}

Update() {
    background(200);

    // Draw the slider
    volumeSlider.draw();

    // Display the current value
    fill(0);
    noStroke();
    textSize(14);
    text("Volume: " + volumeSlider.getValue(), 240, 75);
}
```

- `createSlider(x, y, w, h, minVal, maxVal, defaultVal)` returns a `Slider` object.
- `slider.draw()` draws the track and thumb each frame.
- `slider.getValue()` retrieves its current integer value.
- `slider.onChange(function(int newValue))` registers a callback that fires whenever the thumb moves.

Text Input and Checkboxes

```
TextField nameField;
Checkbox agreeCheck;

Start() {
    createWindow(450, 200, "Form Demo");
```

```
    // TextField at (20, 30), size 200×30, placeholder "Enter name"
    nameField = createTextField(20, 30, 200, 30, "Enter name");

    // Checkbox at (20, 80), size 20×20, label "I agree to the terms"
    agreeCheck = createCheckbox(20, 80, 20, 20, "I agree to the terms");

    // When the user presses Enter inside the text field, print a welcome message
    nameField.onEnter([](String text) {
        println("Hello, " + text + "!");
    });
}

Update() {
    background(245);

    // Draw UI elements
    nameField.draw();
    agreeCheck.draw();

    // If checkbox is checked, show a confirmation
    if (agreeCheck.isChecked()) {
        fill(0, 150, 0);
        textSize(14);
        text("Thank you for agreeing!", 20, 120);
    }
}
```

- `createTextField(x, y, w, h, placeholder)` returns a `TextField`, which accepts keyboard input; pressing Enter will fire `onEnter`.
- `TextField.onEnter(function(String currentText))` is called when the user presses Enter.
- `createCheckbox(x, y, w, h, label)` returns a `Checkbox`.
- `checkbox.draw()` and `checkbox.isChecked()` to read state.
- You can also find `createDropdown()`, `createRadioButtonGroup()`, and other widgets in the Glaze reference if you need more controls.

Handling Low-Level Events

If you need to detect raw keyboard or mouse events (outside of widgets), Glaze provides the familiar functions:

- `keyPressed()`, `keyReleased()`, and you can inspect `key` (char) or `keyCode` (int) inside those handlers.
- `mousePressed()`, `mouseReleased()`, `mouseMoved()`, and check `mouseX`, `mouseY` global variables.
- For more advanced use (e.g., tracking two mouse buttons or gamepad input), you can call `getSystemEvent()` in `Update()` to fetch SFML's `Event` object and inspect it directly. However, the built-in wrappers suffice for most sketches.

6.3 Summary

In this chapter, we introduced two canonical frameworks that demonstrate how Divooka can be used in different contexts. **Slide Present** represents the dataflow-oriented side of Divooka, while **Glaze!** showcases a procedural, sketch-based approach. Both frameworks leverage Divooka's core strengths - graph-based asset organization and cross-platform execution - yet they serve distinct purposes and use cases.

- **Slide Present** lets you build a complete presentation as a single Divooka graph. You define a top-level Presentation, group slides into Slide Sections, and customize each Slide's layout with built-in presets (Hero, One

Body, Two Body, Multi-Body). Since assets (images, audio, video) live alongside your presentation source files, changes appear immediately when you run the graph. Slide Present also supports embedding Divooka subgraphs into slides, so you can include interactive visual demos or dynamically generated media. When you're ready to share, you can export your presentation to Markdown, PDF, or PowerPoint - keeping in mind that only supported content (text, static images) will translate cleanly into those formats.

- **Glaze!** provides a Processing-inspired environment for writing interactive sketches inside Divooka. You start by calling `createWindow()` in `setup()`, then use a familiar `Update()` loop to render 2D shapes (`rect()`, `ellipse()`, `line()`), apply transformations (`pushMatrix()`, `rotate()`), and animate objects. Glaze is built on SFML, so it also handles loading and displaying images (`loadImage()`/`drawImage()`), playing video clips (`loadVideo()`/`drawVideo()`), and streaming audio (`loadSound()`, `play()`, `setLoop()`). For simple 3D scenes, you switch into 3D mode with `begin3D()`, configure a camera, and draw primitives like `box()`, `sphere()`, or custom meshes. Beyond rendering, Glaze extends Processing's model by including first-class GUI widgets - buttons, sliders, text fields, checkboxes - each with callback hooks (`onClick()`, `onChange()`, `onEnter()`). That means you can prototype small tools or demos with interactive controls without pulling in an external UI library.

Together, these two frameworks illustrate how Divooka spans different development paradigms:

1. **Dataflow-centric authoring** with Slide Present, ideal for creating shareable slide decks organized as graphs of assets and subgraphs.
2. **Procedural, code-centric sketches** with Glaze!, designed for quick visualization, multimedia demos, and lightweight GUI applications.

As you explore Divooka's capabilities, you can mix and match these approaches. For example, you might embed a Glaze sketch inside a Slide Present slide to demonstrate an interactive data visualization. Or you might use Slide Present's export features to generate documentation around a Glaze prototype. Chapter 6 has given you the building blocks to:

1. Construct a graph-based presentation complete with reusable Slide Sections and asset references.
2. Create procedural sketches that render 2D/3D content, play audio/video, and respond to GUI widget events.

With these patterns in hand, you can start building more complex Divooka applications - whether that's a fully featured slideshow with embedded interactivity or a standalone sketch that combines graphics, sound, and user controls.

Chapter 7. Other Goodies

7.1 Modularization

Congratulations on completing your introductory journey! Very soon when you started creating programs in Divooka, there will be occasions where you do similar things over and over - it's time to modularize and reuse existing code logic!

Modularization refers to finding common patterns of your graph and extract them out as reusable pieces - this process is called **"refactoring"**. In Divooka, this is done with subgraphs, or saving your graph as documents then reference them in other workflows.

To reference existing graphs, use the **Graph Reference** node from **Programming** toolbox.

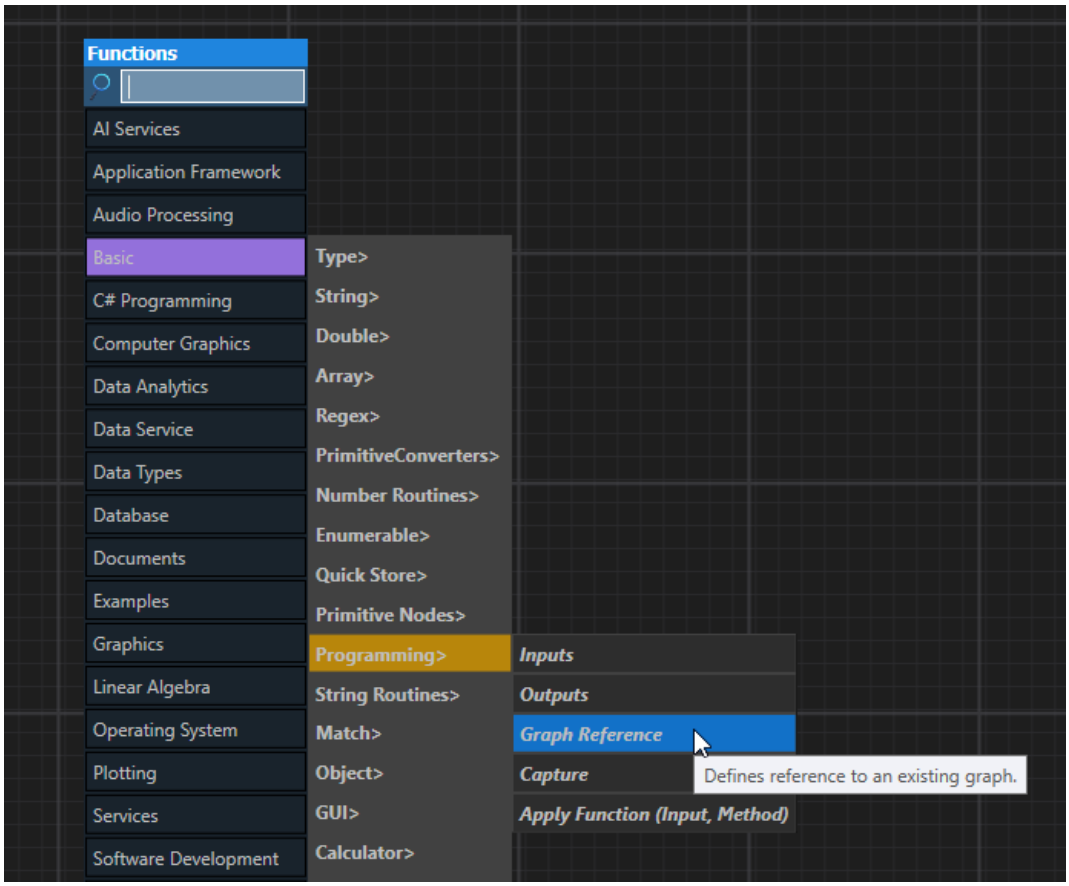


Fig. 7.1 Graph Reference Node from Programming

Inside the graph, we need to define **Inputs** and **Outputs**, which will become connectors on the Graph Reference node.

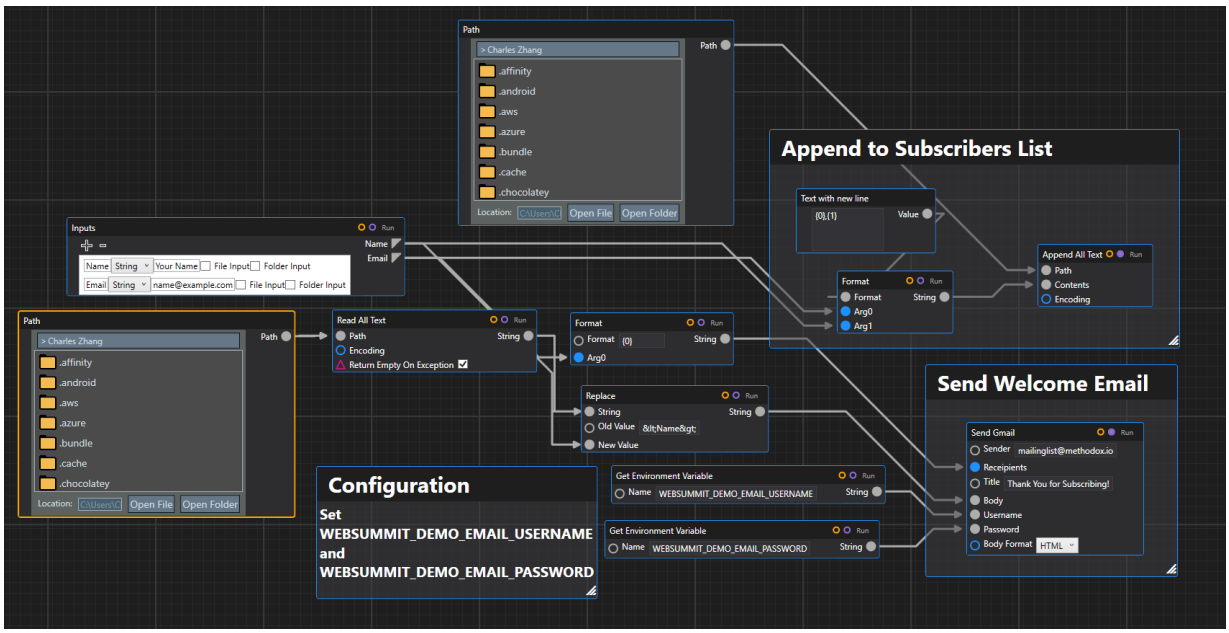


Fig. 7.2 A Reusable Setup Containing Inputs Node

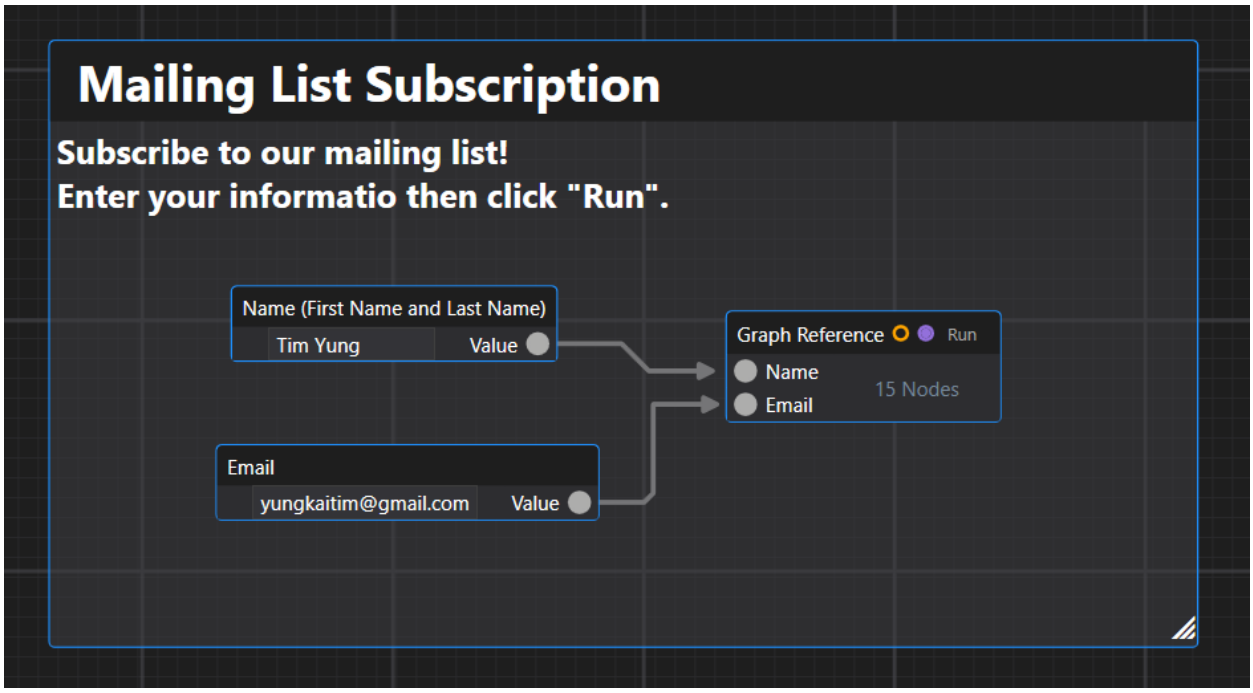


Fig. 7.3 Reuse Existing Component Through Graph Reference

There are many ways you can achieve modularization. Besides saving as a separate document, you can also use ***Subgraphs***.

Subgraphs : A way to organize and reuse existing workflow without needing to save as a separate document.

Subgraphs are embedded directly in the current workflow document.

7.2 Custom GUI, App Mode & Reactive Mode

The Neo graph editor provides the capability to customize interactive graphical user interface application directly on the graph editor, with bookmarks and sidebars. This can be used for simple application GUI and dashboards.

Bookmarks

Sidebars

(Bookmarks, groups, sidebar toggles)

App Mode

The graph editor interface has three different visual modes: Default, Read Only, App Mode.

(Fig 7.x Three Different Visual Modes of the same Group of Nodes)

With the introduction of more customizable primitive nodes, like GUI control nodes and resizable Preview and primitive inputs, one can make simple looking GUIs.

(Fix 7.x Node Arrangements Mimicking a GUI)

When layout nodes, one can hold down `Shift` key to enable snapping for precise placement.

After you are satisfied with the layout, with App Mode on, save the document, and next time when someone opens it it will be in app mode. To avoid accidental modification altogether, the viewers can make use of the `Divooka Viewer` program shipped with `Divooka Explore`.

7.3 Sharing

There are many ways you can share your creations in Divooka.

If you are a plugin or library developer, you can publish your assembly to NuGet.

(Show how to use packages from NuGet)

If you have created a graph, you can send that directly to a friend or colleague, or make use of Divooka Central (you will need a Methodox One account).

(Show screenshot on using it)

7.4 Summary

Where to go from here?

Divooka was started with data analytics in mind. In this short text we have seen how we can use SQL to perform in-memory transformation of tabular data. This topic is worth more extensive discussion (we may share a blog post on this) on rationale and how to debug problems with `InMemorySQLite` module.

We've gone a long way and this is only the start of your Divooka journey. Divooka is set to get everyday programming tasks done and that is a lot! As you read, learn and practice more, the visual node interface is going to become more and more intuitive as your productivity boosts. When the time comes, you might want to mix in some codes and step out of the comfort zone of pure visual flows - thankfully, Divooka got you covered in that as well!

Concepts like regular expressions, logic programming, SQL(lite) language deserve dedicated treatments on their own and is omitted in this introductory book.

Divooka is a vast distribution and feature-rich programming language. In this book we only covered the very basic use cases and the most generic situations. Specialized libraries exist for special tasks: logic programming, NoSQL database, etc. There are also frameworks for specialized graphical applications like interactive media, visual novel, web applications and GUI applications. When it comes to specific program development there is a lot more detail to cover on plugin development, program organization, interface design - the book *Programming with Divooka - Definitive Guide* provides more extensive treatment on those subjects.

The Neo graph editor also has lots of features that's Windows specific but nonetheless very interesting and useful - Neo's reactive mode (App mode) and readonly mode can be used to make quick presentations and even interactive apps. Handy plugins like keyboard display is useful for those making video recordings, and frameworks like Jarvis can make daily life interacting with AI agents easier. We have also not talked in much detail about using multiple graphs in a single document. Readers are encouraged to consult Neo's user manual or our online wiki for that.

When it comes to interfacing with the broader world, Python integration is still in development but already quite useful. To get started, check out Streamlit for Divooka for an easy way to build dashboards.

For production-level application within tightly-integrated development environments, there is a lot to talk about on text-based languages like C#, Pure, Mini Parcel, and CLI tools like Stewer...

For those at the forefront of visual programming, consider subscribing to our Dev Community, Medium and official Methodox Blog, where we routinely post latest updates and tricks. More detailed courses and textbooks are expected to come in the near future.

Thank you for reaching the end of this book, and we hope this book has been instructive in your journey to become a Divooka programmer and see you on the next journey!

Glossary

Assembly

Dataflow Context

Divooka : A visual programming system with IDE, visual driven interface and a large suite of standard libraries.

Divooka Data Grid : A representation of 2D tabular data in Divooka.

Divooka Central : A hub for publishing and sharing everything Divooka.

Document

Extension

Library

Methodox One : A cloud based online service platform by Methodox.

Modularization : Packaging workflows into reusable components that can be latter referenced.

Node : A basic building block in Divooka. Nodes are the visual representation of the functions provided in various toolboxes. Workflows are created by connecting nodes with wires. Connection : Wires between nodes. Connector : Inputs and outputs on the nodes.

Plugin

Procedural Context

Runtime

Subgraph : Embedded workflows in a document.

Tool : Tools are organized under categories in toolboxes. Category : An organization unit of tools. Toolbox : The highest level organization unit of tools on Divooka GUI. Function : Functions are the implementations of nodes.

Workflow

Workspace

Index

[^1] For comprehensive discussion, see https://wiki.methodox.io/en/Development/DivookaExplore [^2] You can read more on online service configuration on https://wiki.methodox.io/en/Divooka/Start/Hello [^3] For a more comprehensive discussion on the topic, see https://wiki.methodox.io/en/Development/DivookaEngine/Plugin