ICT for Civic Data — Crash Course 2026

Visualisation and Presentation

Self-Paced Review — Section F

civicliteraci.es

Why This Matters

> Turn data into visual arguments

Charts, maps, and a portal that communicate clearly to a funder. This is the **Present** step, and it is central to the course. The entire pipeline leads here.

Present is not decoration. It is **communication**: translating data work into something a stakeholder can use. A proposal with a working data portal projects competence in a way that text alone cannot.

The goal of this section: build the visual layer of your proposal: charts that support your argument, and a landing page that ties them together.

Before You Start

> Choosing a chart by task

Your goal	Best chart type	Why
Show change over **time**	Line chart	Continuous progression
Compare categories	Bar chart	Easy side-by-side reading
Show **parts of a whole**	Pie chart (sparingly)	Only with 3-5 categories
Show **distribution**	Histogram	Reveals spread and outliers
Show **relationship**	Scatter plot	Two variables against each other
Show **geographic** patterns	Map	Spatial context matters

The **bar chart is the safe default**. When in doubt, use a bar chart. It is readable, honest, and hard to misinterpret. Pie charts are tempting but misleading above 5 slices because the human eye is bad at comparing angles.

> Storytelling vs exploration

Storytelling

- **Simple**: one message per chart
- **Title states the finding**: "23% of facilities are in flood zones"
- Selective: only show what supports the argument
- Fixed view: the audience sees what you want them to see

Use for: proposals, reports, presentations

Exploration

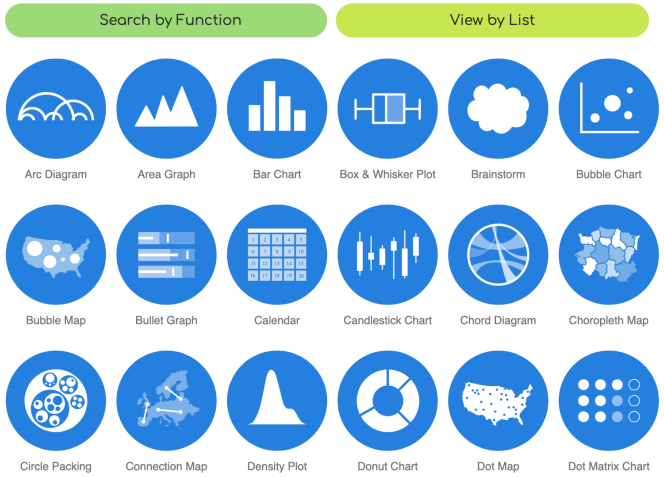
- **Complex**: many variables, interactive filters
- **Title describes the data**: "Health facilities by risk status and region"
- Comprehensive: show everything, let the user find patterns
- Dynamic view: the audience decides what to look at

Use for: data portals, dashboards, internal tools

Your **proposal** is storytelling. Your **data portal** is exploration. Both are valid, but they serve different audiences.

> Data Visualisation Catalogue

datavizcatalogue.com organises chart types by function: comparison, distribution, composition, relationship, hierarchy, flow.



Browse by function when you are not sure which chart type fits. More useful than asking AI because it shows visual examples and trade-offs.

Walkthrough

> Be precise with visualisation prompts

The same precision principle from Section C applies to visualisation. Name the **library**, the **chart types**, and the **layout**.

Vague: "Make me some visualisations of my flood data."

This wastes tokens. The AI will guess chart types, pick a random library, and build something you did not want.

Precise: "Build a Chart.js dashboard with four charts: a bar chart of floods per year, a line chart of average severity over time, a pie chart of causes, and a horizontal bar of top 10 affected regions. Use a card layout."

The AI needs the library name (**Chart.js**, no installation needed, runs in the browser), the specific chart types, and the layout structure. This converges to a useful result on the first try.

> Prompt: build a Chart.js dashboard

« Build a Chart.js dashboard showing analysis of the flood data for [your country]. 1. Read the cleaned CSV file. 2. Create four charts, each in its own card: a bar chart of flood events per year, a line chart of average severity over time, a pie chart of flood causes, and a horizontal bar chart of the top 10 affected regions. 3. Use a responsive card grid layout. A single self-contained HTML file with Chart.js loaded from CDN. Publish it at a different URL path from the map (e.g., /charts/ or /dashboard/). Each chart answers one question. Cards make the layout scannable. A separate URL keeps the dashboard distinct from the map so both can be linked from the landing page. »

Objective | Steps | Output shape | Reasoning

Publish at a **different URL** from the map so you can link to both from the landing page.

> The landing page as a mini data portal

The landing page ties your artifacts together. It is the front door of your data portal:

- **Title and description** — what this portal is about
- **Dataset metadata** — name of each dataset, source, format, date retrieved
- **Navigation** — clear links to the map, the charts, and any other artifacts
- **Context** — one paragraph connecting the data to the proposal's angle

The landing page is where the funder starts. It should take **10 seconds** to understand what the portal contains and how to navigate it. If the funder has to click around to figure out what they are looking at, you have lost them.

> Prompt: build a landing page

« Build a landing page for my data portal about flood risk and health facilities in [your country]. 1. Add a title and one-paragraph description of the project. 2. List the datasets used with metadata: name, source, format, date retrieved. 3. Add navigation cards linking to the map and the charts dashboard. 4. Add a footer with the course name. A single self-contained HTML file as the homepage (index.html). Clean, professional design. Responsive layout. The landing page is the first thing a funder sees. It needs to be clear, informative, and navigable in under 10 seconds. Dataset metadata makes the data pipeline transparent. »

Objective Steps Output shape Reasoning

The landing page becomes the **index.html** of your GitHub Pages site, the URL the funder clicks first.

> v0 by Vercel

v0.dev is an AI-powered webpage creator. It generates polished web interfaces from text descriptions.

Key trick: always ask for **pure HTML/CSS/JS in a single file**. The default output uses React and Next.js, which overcomplexifies the code and makes it hard to modify.

Workflow:

1. Build your HTML first with Gemini or Claude (it works, but looks plain)
2. Upload the HTML to v0 to improve the design
3. Download the result as a single HTML file

Credit warning: v0 charges credits per generation. Students who pasted full CSV datasets into prompts burned all their credits in one session. **Use only sample data** (first 5 rows) in v0 prompts.

> Prompt: clean up the repository

« Clean up the repository structure. 1. Rename files with consistent naming (lowercase, hyphens, no spaces). 2. Ensure the landing page is index.html at the root. 3. Put the map and dashboard in logical subdirectories. 4. Update all internal links. A clean, navigable repo where GitHub Pages serves the landing page at the root URL and all links work. A messy repository makes it hard to find files, breaks links, and looks unprofessional if the funder browses the GitHub repo. »

Objective | Steps | Output shape | Reasoning

Do this **before** the final submission. A clean repository structure is part of the "projecting competence" principle.

> Skills capture reusable processes for AI agents

A **skill** is a markdown file that captures a reusable process. It has:

- **Name** — what the skill does
- **Description** — when to use it
- **Steps** — the procedure, in order
- **Templates** — prompt templates, code snippets

Skills apply across the pipeline: a map-building skill, a data-cleaning skill, a dashboard skill. When you ask the AI to do a task, you tell it to **follow the skill**.

Save AFTER improving the process, not before. The first time you build a map, you learn what works. The second time, you refine. Only then do you save it as a skill.

> Workspace skills apply to one project; global skills apply to all

Workspace scope	Global scope
Saved in the project folder. Applies only to **this project**.	Saved in your user configuration. Applies to **all projects**.
Good for: project-specific conventions, data pipeline steps unique to this dataset, prompts that reference files in this repo.	Good for: general patterns like map building, chart creation, data cleaning steps that work on any dataset.

Keep skills focused. Too many unrelated skills in scope confuse the agent; it may apply the wrong skill to the wrong task. Delete or archive skills you no longer use.

Skills are introduced here because visualisation is where most students first have a process worth saving. But they apply retroactively to everything: finding data, cleaning data, building maps.

Behind the Approach

> Cards structure information for scanning

Each chart or dataset in its own contained box. The human eye processes **grouped, bounded elements** faster than a continuous flow.

Cards work because:

- Each card is **self-contained** — it has a title, a visual, and a brief explanation
- Cards create a **visual hierarchy** — the reader's eye moves from card to card
- Cards are **responsive** — they reflow naturally on different screen sizes

This is the default layout for dashboards, data portals, and even proposal pages. When in doubt about how to present multiple pieces of information, use cards.

> Single-file HTML is simpler for everyone

Self-contained HTML files (one file with HTML, CSS, JS, and data) are the recommended architecture for this course:

- **Easier to move** — copy one file, not a folder tree
- **Easier to test** — open in a browser, no build step
- **Easier for AI** — the agent reads and modifies one file, not ten
- **Privacy-safe** — all data processing happens in the browser, nothing leaves the device

Only very complex, actively developed applications need multi-file architectures. A proposal data portal does not.

When the AI suggests splitting into separate CSS and JS files, say **"keep everything in one HTML file."** This is a deliberate simplicity choice, not a limitation.

> Data privacy with self-contained HTML

Separating data from the interface is not just about architecture; it is about **privacy**:

1. Build the interface with **anonymised sample data** (first 5 rows, randomised coordinates)
2. The AI only needs the **shape of the data**: column names and types, not actual values
3. Self-contained HTML processes data **entirely in the browser**; nothing is sent to a server
4. Connect to real data only after the interface is built and tested

This matters when working with health facility data, population data, or any dataset that could identify individuals or communities. The AI builds with fake data; you verify with real data. The published artifact processes everything client-side.

> The artifact chain

Every step produces a verifiable file. The chain links them:

Prompt → script → commit → CSV → map → chart → portal

If any stone is missing, you cannot retrace the path:

- Lost the script? You cannot reproduce the data extraction
- Skipped the commit? You cannot see what changed between versions
- No CSV? The chart has no verifiable data source

This is why each step is saved, committed, and documented. The artifact chain is your **audit trail**, proof that every result can be traced back to its source.

FAQ

> What if I run out of v0 credits?

Students who pasted **full CSV datasets** into v0 prompts burned all credits in one session. Each generation costs credits, and large inputs consume more.

Workarounds:

- **Use only sample data** — first 5 rows, enough to show the structure
- **Sign up with a different email** — v0 gives credits per account
- **Use Gemini to convert** — if v0 produces React/Next.js output, ask Gemini to rewrite it as pure HTML/CSS/JS in one file

The lesson applies beyond v0: **never paste full datasets into AI tools**. Provide the shape and a small sample. The AI does not need 10,000 rows to build a chart.

> Can I put this on the App Store?

v0 and similar tools create **web applications**, software that runs in a browser and is distributed via a URL.

These are NOT:

- App Store / Google Play apps (those are native apps requiring different development)
- Desktop applications (though a web app can be "installed" as a PWA)
- Offline tools (they need a browser, though self-contained HTML works offline once loaded)

For proposals and data portals, web apps are the right choice: the funder clicks a link and sees the result. No installation, no compatibility issues, works on any device with a browser.

> Can my dashboard update automatically?

A question about whether dashboards can update automatically:

Deterministic automation

Rule-based, reliable. A script runs on a schedule: fetch new data from API → process → update dashboard.

Feasible for: API-driven dashboards with stable data sources.

This works and is appropriate for production systems.

AI-driven automation

AI decides what to fetch, how to process it, and what to show.

Complex, unreliable, and hard to debug when it breaks, which it will.

Warning: tools like OpenClaw are popular but very insecure. They give AI agents unrestricted system access.

For the crash course: manual refresh is fine. For a real project: deterministic automation via scheduled scripts.