DRUPALHELPS HANDBOOK

The Jarvis Handbook

One book for the whole theme. Understand how each part works, build the same thing from an empty folder, then learn every component well enough to hand the site to an editor.

This handbook assumes you know Drupal as a site builder. You have made content types, placed blocks, and edited a view. You do not need theming experience. Every term gets explained the first time it appears, and every code sample is real code from the working theme.

> **How this book is arranged**
>
> Each chapter opens with the explanation: what the part is, why it exists, how it fits with the rest. Then a **Build it yourself** panel marks the hands on half, where the same ground is covered as instructions you can type. The final chapter is a reference to all nineteen components, written for somebody who has never met a Single Directory Component.

Contents

1. The 30,000 foot view

The problem this theme solves

A traditional Drupal theme styles whatever Drupal hands it: node templates, field templates, views templates. The editor gets very little say over layout, and every new design need turns into a developer request.

Jarvis works differently. It ships a library of reusable building blocks (a hero, a card, a stat, a two column layout) and lets editors assemble pages from them visually using Drupal Canvas. Canvas is the drag and drop page builder in Drupal 11. The theme's job becomes three things: define the blocks, style them, and make sure an editor cannot accidentally publish something unreadable.

That last part matters more than it sounds. Give someone a colour picker and an image upload, and eventually you get pale grey text on a bright photo. A good share of this theme exists to prevent exactly that.

The three pieces

The project spans three things across two Git repositories. Knowing which piece owns what saves a lot of confusion later.

Piece	Lives in	Owns
The theme `jarvis`	Its own repo, pulled in as a Git submodule	Components, CSS, JavaScript, template overrides, theme settings

Two custom modules `jarvis_blocks`, `jarvis_canvas`	The recipe repo, under `web/modules/custom`	Behaviour a theme is not allowed to do
The recipe `recipes/jarvis`	The recipe repo	Configuration and demo content. Turns an empty Drupal into a working site

> **Why modules at all?**
>
> A theme's hooks only run while that theme is active. Node edit forms usually render in the admin theme, so a theme hook cannot reach them. Anything that must work regardless of the active theme has to live in a module. That one rule explains most of what these two modules do.

How a page actually renders

Follow one request from start to finish. This is the mental model everything else hangs on.

```
1. Drupal builds the page and fills its regions
   (header, primary_menu, content, footer, and the rest)

2. jarvis.theme runs hook_preprocess_html()
   Reads theme settings, writes CSS custom properties into an inline style tag:
   --bs-primary, --jarvis-fs-h1, --jarvis-bg-1-text ...

3. page.html.twig lays out the regions
   Full width bands, boxed containers, sidebars that collapse when empty

4. Inside the content region, Canvas renders a component tree
   Each entry is a component: jarvis:hero, jarvis:two-column, jarvis:card ...

5. Each component's Twig prints markup and Bootstrap classes,
   reading the CSS custom properties written in step 2

6. Attached JavaScript runs as Drupal behaviors
   contrast.js measures the background image and raises the overlay
   until the text passes WCAG AA
```

Steps 2 and 5 are the pair that matters. Theme settings do not write CSS files. They write *variables*, and the stylesheet reads them. That indirection is what lets a site builder change the whole palette without a developer touching a stylesheet.

The directory map

```
web/themes/custom/jarvis/
├── jarvis.info.yml           Theme definition: regions, libraries, base theme
├── jarvis.libraries.yml      Asset bundles (CSS and JS grouped into libraries)
├── jarvis.theme              PHP: preprocess hooks and shared helpers
├── theme-settings.php        PHP: the admin form for colours, fonts, sizes
├── components/               One folder per Single Directory Component
│   ├── hero/
│   │   ├── hero.component.yml   Schema: props and slots
│   │   ├── hero.twig            Markup
│   │   └── hero.css             Styles, loaded only when the component renders
│   └── card/ text/ stat/ two-column/ ...   (19 in total)
├── css/
│   ├── overrides.css         The main stylesheet
│   ├── ckeditor5.css         Styles the rich text editing area
│   └── canvas-overlay.css    Styles the editor tooling
├── js/
│   ├── wcag.js               The shared contrast engine
│   ├── contrast.js           Live site: auto raises image overlays
│   ├── canvas-overlay.js     Editor: overlay slider and contrast badge
│   └── color-settings.js     Settings form: live contrast badges
├── templates/
│   ├── page.html.twig                  Region layout
│   ├── menu--main.html.twig            Main menu markup
│   ├── jarvis-spacing.html.twig        Shared Twig macro
│   └── jarvis-columns-shell.html.twig  Shared layout wrapper
└── docs/                     This documentation
```

> **Build it yourself**
>
> What follows is the hands on half: the same ground as above, but written as instructions you can type.

1. What we're building

Jarvis is a **base theme + component kit**. It gives a site builder:

- A responsive page shell (header, nav, hero, sidebars, footer) built on Bootstrap 5.
- A palette of drag-and-drop **components** (Hero, Card, Section, Text, Image, Video, 1/2/3-column layouts) that non-developers place visually in **Canvas**.
- A theme settings screen where an admin picks brand **colours** and **fonts** with no code.
- A **recipe** so the whole thing, theme, modules, demo content, installs in one step.

Final folder layout:

```
web/themes/custom/jarvis/
├── jarvis.info.yml          # tells Drupal "I am a theme", declares regions
├── jarvis.libraries.yml     # declares CSS/JS bundles
├── jarvis.theme             # PHP: preprocess hooks (colours, fonts output)
├── theme-settings.php       # PHP: the admin settings form (colour + font pickers)
├── logo.svg
├── templates/
│   └── page.html.twig       # the page shell / region layout
├── css/
│   ├── overrides.css        # Bootstrap variable overrides + spacing utilities
│   ├── color-settings.css   # styles for the settings form
│   └── font-settings.css
├── js/
│   ├── color-settings.js    # settings-form behaviour
│   └── font-settings.js
├── libraries/bootstrap/     # vendored Bootstrap 5 CSS + JS
├── components/              # ← the Single Directory Components
│   ├── hero/  card/  section/  text/  image/  video/
│   ├── one-column/ two-column/ three-column/
│   └── (each has .component.yml + .twig + optional .css/.js)
├── google-fonts.json        # font catalogue for the settings dropdown
└── recipe/                  # the installer (config + demo content)
    ├── recipe.yml
    ├── config/
    └── content/
```

---

2. Key concepts, in plain English

Term	What it means for this tutorial
Theme	The folder that controls how your site *looks*. Jarvis stands alone, it inherits from no other theme.
Region	A named drop-zone in the page (Header, Footer, Content…). Blocks and menus get placed into regions.
Twig	Drupal's templating language (`{{ variable }}`, `{% if %}`). HTML with holes you fill in.
Library	A named bundle of CSS + JS. Drupal only loads a library when something asks for it.
SDC (Single Directory Component)	A self-contained UI widget living in one folder: a `*.component.yml` (its "form"/schema) plus a `*.twig` (its HTML), and optional CSS/JS. Drupal core auto-discovers them.
Props & Slots	A component's inputs. **Props** are simple values (text, a number, a colour). **Slots** are *areas you drop other components into* (like a column that holds cards).
Drupal Canvas	A visual, drag-and-drop page builder (the successor experience to Layout Builder). It reads your SDC components and lets site builders arrange them on a canvas.
Content template	A saved Canvas layout for a *content type*, e.g. "every Article renders its hero field as a Hero component, then its body as Text."
Recipe	A YAML-based installer. It turns on modules, imports config, and loads demo content in one command. Replaces the old "install profile / features" dance.

The mental model: you build *components* once (developer job), then a *site builder* composes pages out of them in Canvas (no code). The theme settings let an admin re-skin the whole thing (colours/fonts) without touching CSS.

---

3. Prerequisites & site setup

You need a working Drupal 11 site. This project uses **DDEV** (Docker-based local dev).

BASH

```
# From an empty project folder:
ddev config --project-type=drupal --docroot=web
ddev start
ddev composer create drupal/recommended-project:^11
ddev composer require drush/drush
ddev drush site:install --account-name=admin --account-pass=admin -y
ddev launch          # opens the site in your browser
```

Jarvis relies on **Drupal Canvas** and a few contrib modules. Install them now so they're available while you build:

BASH

```
ddev composer require drupal/canvas drupal/focal_point drupal/twig_tweak
```

> **Note on Canvas:** Canvas is the drag-and-drop builder. Its machine name is `canvas`, and it ships a companion `canvas_field_component`. You *can* build and preview all the SDC components without Canvas, they also work in Layout Builder, but the Canvas page regions and content templates in Step 9-10 require it.

Your theme goes in `web/themes/custom/jarvis/`. Create that folder:

BASH

```
mkdir -p web/themes/custom/jarvis
cd web/themes/custom/jarvis
```

Everything below is created inside that folder.

Step 1, Scaffold the theme

1a. `jarvis.info.yml`, the theme's identity card

Every theme needs a `*.info.yml`. The filename's prefix (`jarvis`) is the theme's **machine name** and must match the folder name.

YAML

```yaml
name: Jarvis
type: theme
base theme: false
core_version_requirement: ^11 || ^12
description: 'DXPR-style SDC theme with Bootstrap 5, built for Drupal Canvas and La
package: Custom
libraries:
  - jarvis/global
regions:
  header: Header
  primary_menu: 'Primary menu'
  breadcrumb: Breadcrumb
  page_title: 'Page title'
  hero: Hero
  highlighted: Highlighted
  content_top: 'Content top'
  sidebar_left: 'Left sidebar'
  content: Content
  sidebar_right: 'Right sidebar'
  content_bottom: 'Content bottom'
  footer: Footer
```

Line by line:

- **`base theme: false`**, we inherit from no other theme. Drupal falls back to each module's own default templates, so we still get sensible core markup for free and override only what we care about. This is what Olivero and Claro do, and it is the direction core is heading: `stable9` is marked deprecated.
- **`core_version_requirement`**, works on Drupal 11 and 12.

- `libraries: - jarvis/global`, load the `global` library (defined next) on *every* page. That's how Bootstrap gets onto the page.
- `regions:`, the list of drop-zones. The **key** (`header`) is the machine name your Twig uses (`page.header`); the **value** (`Header`) is the human label shown on the Block layout admin screen. These region names must line up with your `page.html.twig` (Step 3).

1b. `jarvis.libraries.yml`, declaring CSS/JS

A **library** is a named bundle. Drupal loads it only when referenced.

YAML

```yaml
global:
  version: 1.x
  css:
    base:
      libraries/bootstrap/css/bootstrap.min.css: { minified: true }
      css/overrides.css: {}
  js:
    libraries/bootstrap/js/bootstrap.bundle.min.js: { minified: true, attributes: {

color-settings:
  version: 1.x
  css:
    theme:
      css/color-settings.css: {}
  js:
    js/color-settings.js: {}
  dependencies:
    - core/once
    - core/drupal

font-settings:
  version: 1.x
  css:
    theme:
      css/font-settings.css: {}
  js:
    js/font-settings.js: {}
  dependencies:
    - core/once
```

```
    - core/drupal
    - core/drupalSettings
```

- **`global`** is attached in `info.yml`, so it's everywhere. It loads Bootstrap's CSS/JS plus our `overrides.css`.
- **`css` groups** (`base`, `theme`...) control load order/weight. `base` loads before `theme`, so component CSS can win over Bootstrap.
- **`color-settings`** / **`font-settings`** are *only* attached to the theme settings admin form (Step 7-8), never on the front end. `dependencies` pull in Drupal's JS helpers (`once`, `Drupal.behaviors`, `drupalSettings`).

1c. A logo and an empty theme file

- Drop a `logo.svg` in the theme root.
- Create an (initially near-empty) `jarvis.theme`, a PHP file for hooks. We fill it in Steps 7-8:

PHP

```php
<?php

/**
 * @file
 * Jarvis theme hooks.
 */
```

> **Clear the cache after any `.yml` change.** Drupal caches discovery aggressively: `ddev drush cr`. Get in the habit, 90% of "my change didn't show up" is a stale cache.

Now enable the theme so you can see progress:

BASH

```
ddev drush theme:enable jarvis
ddev drush config:set system.theme default jarvis -y
ddev drush cr
```

---

2. Single Directory Components

What a component is

A Single Directory Component (SDC) is a Drupal core feature. It is a folder holding everything one interface element needs: a schema file describing its inputs, a Twig template producing its markup, and optionally its own CSS and JavaScript. Put the folder in `components/` and Drupal finds it. There is nothing to register.

The payoff is that the component becomes addressable. Render it from Twig, from PHP, from a Views row template, or from a drag and drop editor. One definition, many callers.

Anatomy

Every component has at least two files, named after the folder.

```
components/text/
├── text.component.yml   The schema. What can go in
├── text.twig            The markup. What comes out
└── text.css             Optional. Loaded only when this component renders
```

Props and slots: the two kinds of input

This distinction trips up nearly everyone at first, so get it straight early.

	Props	Slots
Hold	Values: strings, numbers, booleans, choices, images	Other rendered content, including other components

Declared under	`props: properties:`	`slots:`
Editor sees	A form field in the settings tray	A drop target on the page
Good for	A heading, a colour choice, an alignment	The contents of a column

A two column layout has slots, because you drop things into it. A stat card has props, because you type values into it. A card has props only, because its body is a rich text prop rather than a drop target.

Reading a real schema

Here is the Text component, trimmed to the interesting parts.

COMPONENTS/TEXT/TEXT.COMPONENT.YML

```
$schema: https://git.drupalcode.org/project/drupal/-/raw/HEAD/core/assets/schemas/v
name: Text
status: stable
description: 'Rich text block with width and alignment controls.'
category: Jarvis

props:
  type: object
  required:
    - body
  properties:

    body:
      title: Body
      type: string
      # Tells Canvas this is HTML, so it offers a rich text editor
      # instead of a plain textfield.
      contentMediaType: text/html
      'x-formatting-context': block
      examples:
        - '<h2>Section heading</h2><p>Body copy.</p>'

    width:
      title: Width
      type: string
```

```
    # enum gives a select list instead of a free text field.
    enum: [narrow, normal, wide]
    # meta:enum supplies the human labels for those machine values.
    'meta:enum':
      narrow: Narrow
      normal: Normal
      wide: Wide
    default: normal
    examples:
      - normal
```

Four keys do most of the work.

- `required` lists props the component cannot render without. Canvas will not let an editor save an empty one.
- `enum` turns a field into a select list. Always pair it with `meta:enum` so editors see words instead of machine names.
- `default` is used when nothing is supplied.
- `examples` feeds the preview in the component picker.

> **Give image components an example**
>
> An image based component with no `examples` on its image prop previews as an empty box, because there is nothing to draw. Point the example at a real file. Canvas turns a path like `/sites/default/files/2026-07/banner.png` into a working URL.

Writing the Twig

The template receives every prop as a variable. Here is the whole Text component.

COMPONENTS/TEXT/TEXT.TWIG

```
{% import '@jarvis/jarvis-spacing.html.twig' as jarvis %}
{% set space = jarvis.classes(padding|default('none'), padding_size|default('medium
{% set width = width|default('normal') %}
{% set align = align|default('start') %}
{% set col = { narrow: 'col-lg-6', normal: 'col-lg-8', wide: 'col-12' }[width] %}
{% set align_class = { start: 'text-start', center: 'text-center', end: 'text-end'
<div class="jarvis-text container {{ space }}">
  <div class="row justify-content-center">
```

```
    <div class="{{ col }} {{ align_class }}">
      {{ body|raw }}
    </div>
  </div>
</div>
```

Three patterns worth copying.

Map choices to classes with a Twig hash. The line `{ narrow: 'col-lg-6', ... }[width]` is a lookup table. It keeps class names next to the values they belong to, and it fails loudly if you add an enum value and forget to map it.

Guard every optional prop with `|default()`. Drupal can run Twig in strict mode, where reading an undefined variable is a fatal error rather than an empty string. The filter makes the template safe either way.

Use `|raw` only on already filtered HTML. The `body` prop went through Drupal's text format system, which strips dangerous markup, so printing it raw is correct. Never use `|raw` on a plain string an editor typed.

> **A Twig comment inside a tag is not a comment**
>
> Twig comments work between tags, not inside one. This looks harmless and takes the component down with a lexer error:
>
> ```
> {% set map = {
>   none: ['a'],
>   {# explaining the next line #}
>   dark: ['b'],
> } %}
> ```
>
> Put the note above the `{% set %}` instead. This exact mistake broke three components on this project.

Sharing code between components

Nineteen components will repeat themselves. Twig gives you two tools, and they solve different problems.

A macro, for computing values

Every component accepts padding and margin props. Rather than repeat the class building logic nineteen times, one macro owns it.

TEMPLATES/JARVIS-SPACING.HTML.TWIG

```
{% macro classes(padding, padding_size, margin, margin_size, vertical_padding, no_p
  {{- [
    padding != 'none' ? 'jp-p-' ~ padding ~ '-' ~ padding_size : '',
    margin != 'none' ? 'jp-m-' ~ margin ~ '-' ~ margin_size : '',
    vertical_padding ? 'jp-vpad' : '',
    no_padding ? 'jp-nopad' : '',
  ]|join(' ')|trim -}}
{%- endmacro %}
```

Call it with one import and one line. Change spacing behaviour once, and every component follows.

An embed, for sharing markup

The one, two, and three column layouts were originally near identical files. Each carried the same wrapper, background handling, and contrast matrix. Only the column divs differed. That is a job for `{% embed %}`, which pulls in a shared template and lets the caller fill named blocks.

COMPONENTS/TWO-COLUMN/TWO-COLUMN.TWIG

```
{% set cols = {
  even: ['col-12 col-md-6', 'col-12 col-md-6'],
  'wide-left': ['col-12 col-md-8', 'col-12 col-md-4'],
  'wide-right': ['col-12 col-md-4', 'col-12 col-md-8'],
}[ratio|default('even')] %}

{% embed '@jarvis/jarvis-columns-shell.html.twig' with { count: 2 } %}
  {% block columns %}
    <div class="{{ cols[0] }}">{{ first }}</div>
    <div class="{{ cols[1] }}">{{ second }}</div>
  {% endblock %}
{% endembed %}
```

The shell declares the block the caller fills.

`TEMPLATES/JARVIS-COLUMNS-SHELL.HTML.TWIG (ENDING)`

```
<div class="jarvis-columns jarvis-columns--{{ count }} {{ space }} {{ bg_class }}">
  <div class="{{ container }}">
    <div class="row {{ gap_class }}">
      {% block columns %}{% endblock %}
    </div>
  </div>
</div>
```

Notice the embed does not use `only`. That means the shell still sees every variable the caller has, so the shared props resolve without being restated at three call sites. It is deliberate, and it deserves a comment so nobody tidies it away later.

> **The result**
>
> The one column component went from 54 lines to 16. More usefully, a fix to the contrast logic now happens in one file instead of three.

How Canvas sees your components

With Canvas installed, Drupal creates a configuration entity per component, named `canvas.component.sdc.jarvis.<name>`. It records which field widget each prop uses, and a version hash of the schema.

Two consequences show up quickly.

Placed components pin a version. When an editor places a card, the placement records the schema hash at that moment. Change the schema later and existing placements keep the old version until they are re placed or migrated. Canvas keeps old versions inside the config entity so nothing breaks.

Not every schema edit moves the hash. Useful to know before re exporting configuration.

Change	Moves the version hash?
Adding or changing an `enum`, `type`, or `default`	Yes

Adding `examples` to a **prop**	No
Adding a `description` to a prop	No
Adding `examples` to a **slot**	Yes

> **Build it yourself**
>
> What follows is the hands on half: the same ground as above, but written as instructions you can type.

Step 4, Your first Single Directory Component

Now the fun part. An SDC lives in `components/<name>/` and needs at minimum two files:

- `<name>.component.yml`, the **schema**: its name, category, and the **props** (inputs).
- `<name>.twig`, the **HTML**, which receives those props as variables.

Drupal core discovers any component under `components/` automatically. No registration, no hook. Clear cache and it appears (including inside Canvas's component list).

Let's build the **Text** component, a rich-text block with width and alignment controls.

4a. `components/text/text.component.yml`

YAML

```yaml
$schema: https://git.drupalcode.org/project/drupal/-/raw/HEAD/core/assets/schemas/v
name: Text
status: stable
description: 'Rich text block with width and alignment controls.'
category: Jarvis
props:
  type: object
  required:
    - body
  properties:
    body:
      title: Body
      type: string
      contentMediaType: text/html
```

```
      'x-formatting-context': block
      examples:
        - '<h2>Section heading</h2><p>Body copy with <strong>rich text</strong>.</p
    width:
      title: Width
      type: string
      enum: [narrow, normal, wide]
      'meta:enum':
        narrow: Narrow
        normal: Normal
        wide: Wide
      default: normal
    align:
      title: Alignment
      type: string
      enum: [start, center, end]
      'meta:enum':
        start: Left
        center: Center
        end: Right
      default: start
    padding:
      title: Padding
      type: string
      enum: [none, all, top, bottom, left, right]
      'meta:enum': { none: None, all: 'All sides', top: Top, bottom: Bottom, left:
      default: none
    padding_size:
      title: 'Padding size'
      type: string
      enum: [small, medium, large]
      'meta:enum': { small: 'Small (10px)', medium: 'Medium (20px)', large: 'Large
      default: medium
    vertical_padding:
      title: 'Vertical padding (30px)'
      type: boolean
      default: false
```

How to read the schema, this is the most important thing in the whole tutorial:

- `$schema` points at Drupal's SDC metadata schema. It gives you editor validation and is required for the component to be "recognised."

- **`name`** / **`category`** show up in Canvas's component picker. Everything with `category: Jarvis` groups together in the sidebar.
- **`status: stable`** marks it production-ready (vs `experimental` / `deprecated`).
- **`props.properties`**, each entry is one input the site builder fills in:
  - **`type`** is a JSON-Schema type (`string`, `integer`, `boolean`, `object`).
  - **`title`** is the field label in Canvas.
  - **`enum`** + **`meta:enum`** turns a string into a **dropdown**: `enum` is the stored values, `meta:enum` is the friendly labels. This is how you give a builder a "Left / Center / Right" picker.
  - **`default`** is the value used when the builder leaves it alone.
  - **`contentMediaType: text/html`** + **`x-formatting-context: block`** tells Canvas "this is a rich-text/WYSIWYG field," not a plain string.
  - **`required`** (top-level list) marks props that *must* have a value. **This matters for Canvas data binding:** a required SDC prop can be bound to a content field, but only if that field is itself required. If a prop won't show the 🔗 "link to field" icon in Canvas, it's usually because the prop is required but the target field isn't.

4b. `components/text/text.twig`

The Twig receives every prop as a top-level variable of the same name.

TWIG

```
{#
/**
 * @file
 * Jarvis text block SDC.
 */
#}
{% import '@jarvis/jarvis-spacing.html.twig' as jarvis %}
{% set space = jarvis.classes(padding|default('none'), padding_size|default('medium
  margin|default('none'), margin_size|default('medium'),
  vertical_padding|default(false), no_padding|default(false)) %}
{% set width_class = { narrow: 'col-lg-6', normal: 'col-lg-8', wide: 'col-lg-10' }[
{% set align_class = { start: 'text-start', center: 'text-center', end: 'text-end'

<div class="jarvis-text {{ space }}">
  <div class="container">
    <div class="row justify-content-center">
      <div class="{{ width_class }} {{ align_class }}">
```

```
        {{ body|raw }}
      </div>
    </div>
  </div>
</div>
```

Notice the **spacing pattern** (`space`): it stitches the `padding` + `padding_size` props into a `jp-p-all-medium`-style class from Step 2c. **Every** Jarvis component reuses this logic. It lives once as a Twig macro in `templates/jarvis-spacing.html.twig` (Drupal exposes each theme's `templates/` dir under the `@jarvis` namespace), and every component imports it, one source of truth is what makes the whole kit feel uniform.

`{{ body|raw }}` prints the HTML without escaping (safe here because Canvas ran it through a text-format filter first).

4c. See it

BASH

```
ddev drush cr
```

Every component's `examples:` values feed the SDC preview, so fill them in, they're your free "Storybook." Or just drop the component into a Canvas page (Step 9).

You've now built a complete component. **Card, Image, and Video follow the exact same recipe**, a `.component.yml` describing props, a `.twig` rendering them, plus a `.css` for component-specific styling. Copy Text, change the props, change the markup.

Step 5, Components with slots (Section & Columns)

Props are *values*. **Slots** are *areas that hold other components*. A slot is how you build a column layout: the builder drops a Card *into* the left column of a 2-column component.

5a. Section, a single-slot wrapper

`components/section/section.component.yml` (trimmed to essentials):

YAML

```yaml
$schema: https://git.drupalcode.org/project/drupal/-/raw/HEAD/core/assets/schemas/v
name: Section
status: stable
description: 'Container/section wrapper that holds other components in its slot.'
category: Jarvis
props:
  type: object
  properties:
    width:
      title: Width
      type: string
      enum: [normal, full]
      'meta:enum': { normal: 'Contained', full: 'Full width' }
      default: normal
    bg:
      title: Background
      type: string
      enum: [none, light, dark, primary, body-tertiary]
      'meta:enum': { none: None, light: Light, dark: Dark, primary: Primary, body-t
      default: none
    # …padding / margin / vertical_padding props, same as Text…
slots:
  content:
    title: Content
    description: 'Components placed inside this section.'
```

The new part is the top-level **`slots:`** key. `content` is a named drop-zone. In the Twig, a slot prints just like a variable:

TWIG

```twig
{% set container = (width|default('normal')) == 'full' ? 'container-fluid' : 'conta
{% set bg_class = bg != 'none' ? 'bg-' ~ bg : '' %}
<section class="jarvis-section {{ bg_class }}">
  <div class="{{ container }}">
    {{ content }}   {# ← whatever the builder dropped into the slot renders here #}
  </div>
</section>
```

5b. Two-column, a multi-slot layout

`components/two-column/two-column.component.yml` declares **two** slots and layout props:

YAML

```yaml
name: '2 Columns'
category: 'Jarvis Layout'
props:
  type: object
  properties:
    width:
      title: 'Section width'
      type: string
      enum: [boxed, full]
      'meta:enum': { boxed: 'Boxed (centred)', full: 'Full width' }
      default: boxed
    gap:
      title: 'Column gap'
      type: string
      enum: [none, small, normal, large]
      'meta:enum': { none: None, small: Small, normal: Normal, large: Large }
      default: normal
    ratio:
      title: 'Column ratio'
      type: string
      enum: [even, wide-left, wide-right]
      'meta:enum': { even: 'Even (1:1)', wide-left: 'Wide left (2:1)', wide-right:
      default: even
    # …padding / margin props…
slots:
  first:
    title: 'Column 1'
    description: 'Left / top column.'
  second:
    title: 'Column 2'
    description: 'Right / bottom column.'
```

And the Twig maps the `ratio` / `gap` props onto Bootstrap's grid classes:

TWIG

```
{% set gap_class = { none: 'g-0', small: 'g-2', normal: 'g-4', large: 'g-5' }[gap|d
{% set cols = {
  even:         ['col-12 col-md-6', 'col-12 col-md-6'],
  'wide-left':  ['col-12 col-md-8', 'col-12 col-md-4'],
  'wide-right': ['col-12 col-md-4', 'col-12 col-md-8'],
}[ratio|default('even')] %}
{% set container = (width|default('boxed')) == 'full' ? 'container-fluid' : 'contai
<div class="jarvis-columns jarvis-columns--2">
  <div class="{{ container }}">
    <div class="row {{ gap_class }}">
      <div class="{{ cols[0] }}">{{ first }}</div>
      <div class="{{ cols[1] }}">{{ second }}</div>
    </div>
  </div>
</div>
```

`col-12` on mobile means each column is full width and they **stack**; `col-md-*` kicks in at ≥768px and puts them **side by side**. That single line is your entire responsive story. `one-column` and `three-column` are the same idea with one / three slots.

Step 6, A component with an image (Hero)

The Hero shows how Canvas passes an **image** into a component. The magic is one line in the schema:

YAML

```
image:
  title: 'Background image'
  $ref: json-schema-definitions://canvas.module/image
```

`$ref: json-schema-definitions://canvas.module/image` is a Canvas-provided prop type. It gives the builder a proper media/image picker, and inside Twig the `image` prop arrives as an object with `image.src`, `image.alt`, `image.width`, `image.height`. You don't build an upload widget, Canvas hands you the resolved values.

`components/hero/hero.component.yml` (key props):

YAML

```yaml
name: Hero
category: Jarvis
props:
  type: object
  required:
    - heading
  properties:
    heading:      { title: Heading, type: string }
    subheading:   { title: Subheading, type: string }
    cta_text:     { title: 'CTA text', type: string }
    cta_href:     { title: 'CTA link', type: string, format: uri-reference }
    image:        { title: 'Background image', $ref: 'json-schema-definitions://can
    overlay:      { title: 'Overlay opacity (%)', type: integer, default: 40 }
    align:
      title: 'Text alignment'
      type: string
      enum: [start, center, end]
      'meta:enum': { start: Left, center: Center, end: Right }
      default: center
    text_color:
      title: 'Text color'
      type: string
      enum: [light, dark]
      'meta:enum': { light: 'Light (#ffffff)', dark: 'Dark (#2d6cdf)' }
      default: light
    # …padding / margin / vertical_padding…
```

`components/hero/hero.twig` consumes the image object:

TWIG

```twig
{% set align_class = { start: 'text-start', center: 'text-center', end: 'text-end'
{% set justify_class = { start: 'justify-content-start', center: 'justify-content-c
{% set has_image = image.src is defined and image.src %}
{% set overlay = overlay|default(40) %}
{% set text_color = text_color|default('light') %}

<section class="jarvis-hero jarvis-hero--text-{{ text_color }} position-relative d-
         {% if has_image %}style="background-image: url('{{ image.src }}');"{% endi
  {% if has_image %}
    <div class="jarvis-hero__overlay" style="opacity: {{ overlay / 100 }};" aria-hi
```

```
  {% endif %}
  <div class="container position-relative jarvis-hero__inner py-5">
    <div class="row {{ justify_class }}">
      <div class="col-lg-9 col-xl-8">
        {% if heading %}<h1 class="jarvis-hero__heading display-3 fw-bold mb-3">{{
        {% if subheading %}<p class="jarvis-hero__subheading lead mb-4">{{ subheadi
        {% if cta_text and cta_href %}<a href="{{ cta_href }}" class="btn btn-prima
      </div>
    </div>
  </div>
</section>
```

Optional polish: a small script can measure the background and *raise* the overlay opacity when white text would fail WCAG AA contrast, the builder sets a floor, the JS enforces legibility. In Jarvis that engine is shared by the hero and card components: it lives in `js/contrast.js`, registered as the `jarvis/contrast` library and attached to each component through `libraryOverrides: dependencies:` in its `*.component.yml` (the alternative, SDC auto-loading a same-named JS file per component, would mean duplicating the code).

Image styles inside components

For the `Image` component, note the `image_uri` prop and the **named image style** trick. Canvas gives you `image.src`, but to render a specific Drupal **image style** (say `hero_banner`) you also pass the raw file URI and let Twig build the derivative with `twig_tweak`'s `|image_style` filter. That's why the recipe installs `twig_tweak`, and why image styles like `hero_banner` and `wide` are shipped as config (Step 11). In a content template, use `|image_style` in the SDC, **don't** try to use a Canvas "adapter" for it (adapters are not allowed inside content templates).

---

3. CSS and the theming contract

Bootstrap as the base

Jarvis ships Bootstrap 5 and uses its grid, utilities, and components. That is a deliberate trade. You inherit a well tested responsive grid and a large utility vocabulary. In exchange you accept Bootstrap's conventions and its file size.

Components use Bootstrap classes directly in their Twig: `container`, `row`, `col-lg-8`, `text-center`. The theme's own CSS handles only what Bootstrap does not.

Custom properties are the contract

This is the central idea of the stylesheet, so it is worth slowing down for.

The theme has seven or eight coloured surfaces: a primary band, a dark band, two arbitrary editor chosen backgrounds, a photo background, a light band, and cards that sit on top of any of them. Every one needs its text, headings, and links to stay readable.

The naive approach repeats a block of colour declarations per surface. This theme did that originally. The block was nine lines and appeared seven times.

The better approach: each surface declares only what it *is*, and one shared rule wires it up.

CSS/OVERRIDES.CSS

```
/* One foreground contract for every surface. Each variant declares only what
   it IS, and this rule maps that onto every colour variable the theme and
   Bootstrap read. */
.jarvis-bg--primary,
.jarvis-bg--secondary,
.jarvis-bg--dark,
.jarvis-bg--light,
.jarvis-bg--bg1,
.jarvis-bg--bg2,
.jarvis-bg--image,
.jarvis-columns .jarvis-card:not(.jarvis-card--background) {
  color: var(--jarvis-on);
  --bs-body-color: var(--jarvis-on);
  --bs-body-color-rgb: var(--jarvis-on-rgb);
```

```
  --bs-heading-color: var(--jarvis-on);
  --jarvis-heading-color: var(--jarvis-on);
  --bs-emphasis-color: var(--jarvis-on);
  --bs-link-color: var(--jarvis-link, var(--jarvis-on));
  --bs-link-color-rgb: var(--jarvis-link-rgb, var(--jarvis-on-rgb));
  --bs-link-hover-color: var(--jarvis-link-hover, var(--jarvis-on));
}

/* Now each surface is three or four lines. */
.jarvis-bg--primary {
  background: var(--bs-primary, #2d6cdf);
  --jarvis-on: #fff;
  --jarvis-on-rgb: 255, 255, 255;
  --jarvis-link-hover: #e6e6e6;
}

.jarvis-bg--light {
  background: #f5f5f5;
  --jarvis-on: #000;
  --jarvis-on-rgb: 0, 0, 0;
  --jarvis-link-hover: #333333;
}
```

Two details make this work.

Custom properties resolve per element. A card inside a dark section inherits `--jarvis-on: #fff`, but the card's own rule sets `--jarvis-on: #1a1a1a` on itself, and a declaration that applies to the element beats an inherited one. The card gets dark text on its light surface with no override gymnastics.

Setting Bootstrap's variables steers Bootstrap's components. Assigning `--bs-body-color` and its siblings makes every Bootstrap element inside the section follow along, without targeting each one.

From theme settings to CSS variables

The admin form does not write a stylesheet. It writes configuration, and a preprocess hook turns that into variables in an inline style tag.

`JARVIS.THEME`

```
function jarvis_preprocess_html(array &$variables) {
  $lines = [];
  foreach (_jarvis_colors() as $key => $info) {
    [, $var, $rgb] = $info;
    $val = theme_get_setting($key);
    // Only emit a valid hex. This closes CSS injection through a setting.
    if (!is_string($val) || !preg_match('/^#[0-9a-fA-F]{6}$/', $val)) {
      continue;
    }
    $lines[] = "$var:$val;";
    if ($rgb) {
      $r = hexdec(substr($val, 1, 2));
      $g = hexdec(substr($val, 3, 2));
      $b = hexdec(substr($val, 5, 2));
      $lines[] = "{$var}-rgb:$r,$g,$b;";
    }
  }

  $variables['#attached']['html_head'][] = [
    [
      '#tag' => 'style',
      // html:root beats a plain :root in overrides.css regardless of order.
      '#value' => 'html:root{' . implode('', $lines) . '}',
    ],
    'jarvis-colors',
  ];
}
```

Three things to copy from this.

- **Validate before emitting.** A setting goes straight into a stylesheet. Without the hex check, a stored value could close the style block and inject anything.
- **Use `html:root`, not `:root`.** One more point of specificity, so it wins over the defaults in the stylesheet no matter which loads first.
- **One list, two consumers.** `_jarvis_colors()` is shared by the settings form and the output, so the two cannot drift apart.

Computing a safe text colour

Two background colours are whatever the site builder picked. The theme cannot hardcode text colour against an unknown background, so it calculates one.

JARVIS.THEME

```
/**
 * Relative luminance of a '#rrggbb' colour, per WCAG 2.x.
 */
function _jarvis_luminance(string $hex): float {
  $c = [];
  foreach ([1, 3, 5] as $i) {
    $v = hexdec(substr($hex, $i, 2)) / 255;
    $c[] = $v <= 0.03928 ? $v / 12.92 : (($v + 0.055) / 1.055) ** 2.4;
  }
  return 0.2126 * $c[0] + 0.7152 * $c[1] + 0.0722 * $c[2];
}

function _jarvis_contrast_ratio(string $a, string $b): float {
  $l1 = _jarvis_luminance($a);
  $l2 = _jarvis_luminance($b);
  return (max($l1, $l2) + 0.05) / (min($l1, $l2) + 0.05);
}

// In the preprocess hook, pick whichever wins:
$light = _jarvis_contrast_ratio($val, '#ffffff') >= _jarvis_contrast_ratio($val, '#
$lines[] = "{$var}-text:" . ($light ? '#fff' : '#000') . ';';
```

The CSS then reads `--jarvis-bg-1-text` and never needs to know what the background colour is.

Styling the editing area

Authors write in CKEditor 5. By default that editing area inherits the admin theme's typography, so what they type looks nothing like what publishes. The fix is one key in the info file plus a stylesheet.

JARVIS.INFO.YML

```
ckeditor5-stylesheets:
  - css/ckeditor5.css
```

Two traps here, both of which cost real time on this project.

Core does not scope the file for you

CKEditor 5 in Drupal is inline, not an iframe, and core loads your stylesheet as a plain library. An unscoped rule restyles the whole admin page. Scope every rule to `.ck-content`.

CKEditor's own styles land after yours

CKEditor injects around seventy `.ck-content` rules as an inline style tag, which comes later in the document. At equal specificity theirs win, so your base font, blockquote, and pre rules get silently overridden. Doubling the class beats them without reaching for `!important`:

```
.ck-content.ck-content blockquote {
  border-inline-start: 4px solid var(--bs-primary, #2d6cdf);
}
```

Variables are missing in the editor

Node forms render in the admin theme, so `hook_preprocess_html()` never runs and the `--jarvis-*` variables are absent. Give every value a fallback, and if you want the editor to track configured sizes, emit them from a *module* hook scoped to `.ck-content`. Convert `rem` to `em` when you do, because `rem` resolves against the admin theme's root, which you must not touch.

Build it yourself

What follows is the hands on half: the same ground as above, but written as instructions you can type.

Step 2, Add Bootstrap and the CSS layer

2a. Vendor Bootstrap

Download Bootstrap 5's compiled dist and place two files:

```
libraries/bootstrap/css/bootstrap.min.css
libraries/bootstrap/js/bootstrap.bundle.min.js
```

(You can grab these from getbootstrap.com → "Download" → `dist/` folder. The `.bundle` JS includes Popper, which the responsive nav needs.)

2b. `css/overrides.css`, theme by *variables*, not by rewriting

Bootstrap 5 is built on **CSS custom properties** (`--bs-primary`, etc.). Instead of forking Bootstrap's SCSS, we just override its variables. This is the whole styling philosophy of Jarvis, *theme at the variable level*.

`css`

```
/**
 * Jarvis theme overrides — DXPR-style look via Bootstrap 5 CSS custom properties.
 */
:root {
  --bs-primary: #2d6cdf;
  --bs-primary-rgb: 45, 108, 223;
  --bs-secondary: #1b2a4a;
  --bs-body-font-family: "Inter", system-ui, -apple-system, "Segoe UI", Roboto, san
  --bs-body-color: #1b2130;
  --bs-border-radius: 0.65rem;
  --bs-border-radius-lg: 1rem;
  --jarvis-section-gap: 4rem;
}

.btn-primary {
  --bs-btn-bg: var(--bs-primary);
  --bs-btn-border-color: var(--bs-primary);
  --bs-btn-hover-bg: #245ac0;
  --bs-btn-hover-border-color: #245ac0;
  --bs-btn-padding-x: 1.6rem;
  --bs-btn-padding-y: 0.7rem;
  font-weight: 600;
}
```

Because Jarvis is a **Canvas SDC theme**, avoid viewport units (`vh` / `vw`), they break the Canvas preview iframe, which has no real viewport. Stick to `rem` / `px` / `%`.

2c. Spacing utilities every component shares

The design calls for exact `10 / 20 / 30px` spacing, which Bootstrap's `rem` scale can't hit. So Jarvis ships its own tiny utility set (prefix `jp-` = "Jarvis padding"). Every component's `padding` / `margin` props compile down to these classes:

CSS

```
/* small=10px, medium=20px, large=30px */
.jp-p-all-small { padding: 10px; }
.jp-p-all-medium { padding: 20px; }
.jp-p-all-large { padding: 30px; }
.jp-p-top-small { padding-top: 10px; }
/* …top/bottom/left/right × small/medium/large, for padding (jp-p-) and margin (jp-

.jp-vpad { padding-top: 30px; padding-bottom: 30px; }
.jp-nopad { padding-top: 0 !important; padding-bottom: 0 !important; }
```

The naming pattern is deliberate: `jp-{p|m}-{side}-{size}`. In Step 4 you'll see the Twig that builds these class names from a component's props, write the pattern once, reuse it in every component.

Step 7, Colour settings

Goal: an admin picks brand colours on the theme settings page, and the whole site re-skins, no CSS edits. The trick: **write the chosen colours out as CSS custom properties**, and have `overrides.css` consume those properties.

Three moving parts:

1. A **shared list** of colour slots (so the form and the output never drift).
2. A **settings form** (`theme-settings.php`) with a hex field + native colour picker.

3. A **preprocess hook** (`jarvis.theme`) that prints the chosen colours as an inline `<style>`.

7a. The shared colour map (`jarvis.theme`)

PHP

```php
/**
 * setting key => [label, CSS custom property, needs -rgb, default].
 */
function _jarvis_colors() {
  return [
    'jarvis_color_primary'     => ['Primary', '--bs-primary', TRUE, '#2d6cdf'],
    'jarvis_color_secondary'   => ['Secondary', '--bs-secondary', FALSE, '#1b2a4a']
    'jarvis_color_title_bg'    => ['Title background', '--jarvis-title-bg', FALSE,
    'jarvis_color_title_text'  => ['Title text', '--jarvis-title-color', FALSE, '']
    'jarvis_color_footer_bg'   => ['Footer background', '--jarvis-footer-bg', FALSE
    'jarvis_color_footer_text' => ['Footer text', '--jarvis-footer-color', FALSE, '
    'jarvis_color_heading'     => ['Heading color', '--jarvis-heading-color', FALSE
    'jarvis_color_header_bg'   => ['Header block region background', '--jarvis-head
    'jarvis_color_header_text' => ['Header block region text', '--jarvis-header-col
    'jarvis_color_header_link' => ['Header block region link color', '--jarvis-head
  ];
}
```

One function, used by both the form and the output, that's what keeps them in sync. The third element (`TRUE`) means "also emit an `-rgb` triplet," which Bootstrap needs for `rgba()` transparency (`--bs-primary-rgb`).

7b. The settings form (`theme-settings.php`)

Drupal automatically calls `hook_form_system_theme_settings_alter()` in a theme's `theme-settings.php` to add fields to the appearance settings page.

PHP

```php
<?php

use Drupal\Core\Form\FormStateInterface;
use Drupal\Core\Render\Markup;

function jarvis_form_system_theme_settings_alter(array &$form, FormStateInterface $
  $form['jarvis_colors'] = [
```

```
    '#type' => 'details',
    '#title' => t('Colors'),
    '#open' => TRUE,
    '#weight' => -20,
  ];

  foreach (_jarvis_colors() as $key => $info) {
    [$label, , , $default] = $info;
    $val = theme_get_setting($key);
    $val = (is_string($val) && $val !== '') ? $val : $default;
    $swatch = ($val !== '') ? $val : '#ffffff';

    $form['jarvis_colors'][$key] = [
      '#type' => 'textfield',
      '#title' => t($label),
      '#default_value' => $val,
      '#size' => 10,
      '#maxlength' => 7,
      '#attributes' => [
        'class' => ['jarvis-color-hex'],
        'pattern' => '#[0-9a-fA-F]{6}',
        'placeholder' => '#ffffff',
      ],
      // A native colour picker as the field prefix; JS keeps it in sync with the t
      '#field_prefix' => Markup::create(
        '<input type="color" class="jarvis-color-pick" value="'
        . htmlspecialchars($swatch, ENT_QUOTES) . '">'
      ),
    ];
  }

  $form['#attached']['library'][] = 'jarvis/color-settings';
  $form['#validate'][] = 'jarvis_color_settings_validate';
}

/**
 * Reject anything that isn't a #rrggbb hex (or empty). Security: this is what
 * stops someone injecting CSS through a colour value.
 */
function jarvis_color_settings_validate(array &$form, FormStateInterface $form_stat
  foreach (array_keys(_jarvis_colors()) as $key) {
    $val = (string) $form_state->getValue($key);
    if ($val !== '' && !preg_match('/^#[0-9a-fA-F]{6}$/', $val)) {
      $form_state->setErrorByName($key, t('%v is not a valid hex color (use #rrggbb
    }
```

```
  }
}
```

Each colour is a hex **textfield** with a native `<input type="color">` glued on as the prefix. The colour input is just the picker/preview; only the text field actually submits. `css/color-settings.js` keeps the two in sync. **Always validate**, the strict hex regex is what prevents CSS injection through a settings value.

7c. Output the colours (`jarvis.theme` preprocess)

PHP

```php
/**
 * Implements hook_preprocess_html().
 * Emit chosen colours as CSS variables in an inline <style>.
 */
function jarvis_preprocess_html(array &$variables) {
  $lines = [];
  foreach (_jarvis_colors() as $key => $info) {
    [$label, $var, $rgb] = $info;
    $val = theme_get_setting($key);
    // Re-validate on output too — only emit clean #rrggbb values.
    if (!is_string($val) || !preg_match('/^#[0-9a-fA-F]{6}$/', $val)) {
      continue;
    }
    $lines[] = "$var:$val;";
    if ($rgb) {
      $r = hexdec(substr($val, 1, 2));
      $g = hexdec(substr($val, 3, 2));
      $b = hexdec(substr($val, 5, 2));
      $lines[] = "{$var}-rgb:$r,$g,$b;";
    }
  }
  if ($lines) {
    $variables['#attached']['html_head'][] = [
      ['#tag' => 'style', '#value' => 'html:root{' . implode('', $lines) . '}'],
      'jarvis-colors',
    ];
  }
}
```

Why `html:root` instead of plain `:root`? Specificity. `html:root` (0,0,1,1) beats `overrides.css`'s `:root` defaults regardless of stylesheet order, so the admin's choices always win. Now, in `overrides.css`, any region that reads one of those variables re-skins itself automatically:

CSS

```
.jarvis-footer { background: var(--jarvis-footer-bg); color: var(--jarvis-footer-co
.jarvis-page-title { background: var(--jarvis-title-bg); color: var(--jarvis-title-
h1, h2, h3 { color: var(--jarvis-heading-color); }
```

That's the payoff: **define a variable in the map → consume it in CSS → the region is themeable with zero extra PHP.**

Step 8, Font settings (self-hosted Google Fonts)

Same idea, one level fancier. The admin picks a Google font per "slot" (base text, headings, nav, site name, blockquote) and targets it with a CSS selector. On **save**, the theme *downloads* those fonts and self-hosts them, so the live site never calls Google (better privacy + GDPR + speed).

8a. A font catalogue

Ship a `google-fonts.json` listing the families and their available weights:

JSON

```
{
  "families": {
    "Open Sans": ["300", "400", "600", "700", "300i", "400i"],
    "Inter":     ["400", "500", "600", "700"],
    "Playpen Sans": ["300", "400", "500"]
  }
}
```

The settings form reads this to populate the family dropdown and per-family weight options.

8b. Font slots (shared map, in `jarvis.theme`)

PHP

```
/**
 * setting-key stem => [label, default CSS selector].
 * Each slot stores three settings: _family, _weight, _selector.
 */
function _jarvis_fonts() {
  return [
    'base'       => ['Base font', 'body'],
    'headings'   => ['Headings', 'h1, h2, h3, h4, h5, h6, .page-title'],
    'navigation' => ['Navigation', 'nav, nav ul li, nav a'],
    'site_name'  => ['Site name', '.site-name, .navbar-brand'],
    'blockquote' => ['Blockquote', 'blockquote, blockquote p'],
  ];
}
```

8c. The form (added to the same `theme_settings_alter`)

For each slot: a **family** `select`, a **weight/style** `select`, and a **CSS selector** textfield, plus a live preview. `js/font-settings.js` repopulates the weight dropdown when the family changes (using the `drupalSettings` catalogue you attached). The form registers a **submit handler**:

PHP

```
$form['jarvis_fonts']['#attached']['library'][] = 'jarvis/font-settings';
$form['jarvis_fonts']['#attached']['drupalSettings']['jarvisFonts']['families'] = $
$form['#submit'][] = 'jarvis_font_settings_submit';
```

8d. The download-and-self-host submit handler

On save, for each chosen font: fetch the Google `css2` payload with a Chrome User-Agent (so Google returns modern `woff2`), download each referenced `.woff2` into `public://jarvis-fonts/`, rewrite the `url()` to the local copy, and append a CSS rule that applies the family to the slot's selector. Everything is written to one `jarvis-fonts.css`:

PHP

```
function jarvis_font_settings_submit(array &$form, FormStateInterface $form_state)
  $fs = \Drupal::service('file_system');
  $dir = 'public://jarvis-fonts';
  $fs->prepareDirectory($dir, /* CREATE_DIRECTORY | MODIFY_PERMISSIONS */ 3);
  $client = \Drupal::httpClient();
  $ua = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) … Chrome/120.0 Safari/537.36';
  // for each slot: build the css2 URL, fetch CSS, download every woff2,
  //   rewrite url() → local, and add a `selector { font-family: … }` rule …
  // then: $fs->saveData($generatedCss, "$dir/jarvis-fonts.css", FileExists::Replac
  \Drupal::messenger()->addStatus(t('Fonts downloaded and self-hosted.'));
}
```

8e. Attach the generated stylesheet on every page

Back in `jarvis_preprocess_html()`, link the generated file if it exists (cache-busted by its mtime):

PHP

```
$css = 'public://jarvis-fonts/jarvis-fonts.css';
$real = \Drupal::service('file_system')->realpath($css);
if ($real && file_exists($real)) {
  $url = \Drupal::service('file_url_generator')->generateString($css);
  $variables['#attached']['html_head'][] = [
    ['#tag' => 'link', '#attributes' => ['rel' => 'stylesheet', 'href' => $url . '?
    'jarvis-fonts',
  ];
}
```

The takeaway pattern is identical to colours: **shared map → form → save/output.** Colours output inline on every request; fonts are generated once at save time into a file that's then attached.

---

4. JavaScript and Drupal behaviors

Behaviors, not DOMContentLoaded

Drupal re renders parts of a page constantly: AJAX views pagers, modal dialogs, the Canvas preview. Code that runs once on page load misses all of it.

The correct pattern is a behavior plus `once()`. The behavior runs again on every re render, and `once()` guarantees each element is processed exactly one time.

THE PATTERN

```
(function (Drupal, once) {
  'use strict';

  function enhance(element) {
    // do the work
  }

  Drupal.behaviors.myFeature = {
    attach: function (context) {
      // context is the newly added part of the page, or the whole document
      once('my-feature', '.my-selector', context).forEach(enhance);
    }
  };
})(Drupal, once);
```

Declare the dependencies, or `Drupal` and `once` will be undefined.

JARVIS.LIBRARIES.YML

```
contrast:
  version: 1.0.5
  js:
    js/contrast.js: {}
  dependencies:
    - jarvis/wcag
    - core/once
    - core/drupal
```

> **This is a real bug, not a theoretical one**
>
> The contrast script originally ran once on `DOMContentLoaded` with a manual flag. Any section arriving later, from a views pager or a Canvas refresh, kept the editor's raw overlay value with no contrast tuning at all. Which is exactly the case the safety net exists for.

One engine, several consumers

Three scripts need WCAG contrast maths: the live site, the Canvas editor badge, and the settings form. Two PHP functions need it too.

They started as five copies, and copies drift. One used `Math.pow` and another `**`. One took hex strings, another took channel numbers. The 4.5 threshold was written three different ways.

Now one file owns it and exposes an API.

JS/WCAG.JS (SHAPE)

```
(function (root) {
  'use strict';

  var NEED = 4.5;        // WCAG AA for normal text
  var TEXT = 1;          // luminance of #fff
  var DARK_TEXT = 0.011; // luminance of #212529

  function chan(c) {
    c /= 255;
    return c <= 0.03928 ? c / 12.92 : Math.pow((c + 0.055) / 1.055, 2.4);
  }
  function lum(r, g, b) {
    return 0.2126 * chan(r) + 0.7152 * chan(g) + 0.0722 * chan(b);
  }
  function contrast(l1, l2) {
    return (Math.max(l1, l2) + 0.05) / (Math.min(l1, l2) + 0.05);
  }

  var api = { NEED: NEED, lum: lum, contrast: contrast /* , ... */ };

  if (typeof module !== 'undefined' && module.exports) {
```

```
    module.exports = api;
    selfCheck();       // runs under `node js/wcag.js`
    return;
  }
  root.jarvisWcag = api;
})(typeof window === 'undefined' ? this : window);
```

Consumers take what they need.

JS/CONTRAST.JS

```
(function (Drupal, once, wcag) {
  'use strict';

  var extremeBlock = wcag.extremeBlock;
  var neededAlpha = wcag.neededAlpha;
  var neededAlphaLight = wcag.neededAlphaLight;

  // ... use them ...

})(Drupal, once, window.jarvisWcag);
```

Leave a runnable check behind

The engine carries a self check that runs under Node and is skipped in the browser. No test framework, no configuration, one command.

RUN IT

```
node js/wcag.js
# wcag self-check ok
```

> **Test the real function, not a copy of it**
>
> An earlier version of this check re implemented the maths inside the test block, because the module exported nothing. The assertions passed happily while the real solver was wrong. If your self check cannot reach the actual exported functions, it is checking nothing.

What the contrast script actually does

Worth understanding, because it is the heart of the accessibility work.

1. Find elements that have a background image and an overlay layer.
2. Draw the image into a 16 by 16 canvas and read the pixels.
3. Split that into sixteen 4 by 4 blocks and average each.
4. Pick the worst block: the brightest if the text is white, the darkest if the text is dark.
5. Step the overlay opacity up until the text clears 4.5:1 against that block.

Sampling blocks rather than a whole image average is the point. Text can cross a bright patch that an average would hide. The editor's chosen opacity is a floor. It is never lowered, only raised when the image demands it.

---

5. Template overrides and preprocess

How Drupal picks a template

Drupal looks for templates by name, most specific first. Put a file with the right name in your theme's `templates/` folder and it wins. There is no registration step, but you do need a cache rebuild after adding one.

File name	Overrides
`page.html.twig`	Every page's region layout
`menu--main.html.twig`	Just the main menu
`node--article--teaser.html.twig`	Article nodes in teaser view mode
`block--my-block-id.html.twig`	One specific placed block

Regions and the page template

Regions are declared in the info file. Anything not listed there cannot receive a block.

JARVIS.INFO.YML

```
regions:
  header: Header
  primary_menu: 'Primary menu'
  breadcrumb: Breadcrumb
  help: Help
  page_title: 'Page title'
  hero: Hero
  highlighted: Highlighted
  content_top: 'Content top'
  sidebar_left: 'Left sidebar'
  content: Content
  sidebar_right: 'Right sidebar'
  content_bottom: 'Content bottom'
  footer: Footer
```

The page template decides which regions are full width bands and which are boxed, and collapses sidebars when they hold nothing.

TEMPLATES/PAGE.HTML.TWIG (EXCERPT)

```
{% set has_sidebar = not sidebar_left_empty or not sidebar_right_empty %}
{% if has_sidebar %}
  <div class="container jarvis-region jarvis-main">
    <div class="row g-4">
      {% if not sidebar_left_empty %}
        <aside class="col-12 col-lg-3 jarvis-sidebar" role="complementary">
          {{ sidebar_left_markup|raw }}
        </aside>
      {% endif %}
      {# `col` alone flexes to fill whatever the sidebars leave #}
      <div class="col jarvis-content">{{ page.content }}</div>
    </div>
  </div>
{% else %}
  {# No sidebars: edge to edge, so a full width layout can break out #}
  <div class="jarvis-content jarvis-content--full">{{ page.content }}</div>
{% endif %}
```

Preprocess: preparing variables before Twig

A preprocess hook runs before a template renders and can add or change the variables it receives. It is where PHP work belongs, so templates stay readable.

Those `sidebar_left_empty` flags come from one. Testing emptiness turned out to be harder than expected, and the reason is instructive.

JARVIS.THEME

```
function jarvis_preprocess_page(array &$variables) {
  $renderer = \Drupal::service('renderer');
  foreach (['header', 'hero', 'sidebar_left', 'sidebar_right'] as $region) {
    $markup = !empty($variables['page'][$region])
      ? (string) $renderer->render($variables['page'][$region])
      : '';
    $variables[$region . '_markup'] = $markup;

    // The Canvas editor wraps even an EMPTY region in comment markers plus a
    // bare div. So "empty" means: no text once tags and comments are stripped,
    // AND no visual element. An image only sidebar has no text but is not empty.
    $stripped = preg_replace('/<!--.*?-->/s', '', $markup);
    $variables[$region . '_empty'] = trim(strip_tags($stripped)) === ''
      && !preg_match('/<(img|picture|video|iframe|svg|canvas|embed)\b/i', $stripped
  }
}
```

Without that, a plain `{% if page.hero %}` reads an empty region as full inside the editor, and the hero paints a 420 pixel gradient band over nothing.

A menu template worth studying

Drupal's menu block emits a bare `nav` and `ul` with no classes. To get Bootstrap dropdowns you supply the markup Bootstrap's JavaScript binds to.

TEMPLATES/MENU--MAIN.HTML.TWIG (EXCERPT)

```
{% macro build(items, depth, attributes) %}
  {% import _self as menus %}
  {% if items %}
    {% if depth == 0 %}
      <ul{{ attributes.addClass('jarvis-menu') }}>
    {% else %}
      <ul class="dropdown-menu">
```

```
    {% endif %}
    {% for item in items %}
      {% set has_children = item.below is not empty %}
      {% set is_parent = has_children and depth == 0 %}
      <li{{ item.attributes.addClass(depth == 0 ? 'nav-item' : '', is_parent ? 'dro
        {% if is_parent %}
          {# The link still navigates. A separate split button opens the panel,
             because data-bs-toggle on the link itself swallows the click. #}
          <a href="{{ item.url }}" class="nav-link">{{ item.title }}</a>
          <button type="button" class="nav-link dropdown-toggle dropdown-toggle-spl
                  data-bs-toggle="dropdown" aria-expanded="false">
            <span class="visually-hidden">{{ 'Show submenu for @title'|t({'@title':
          </button>
          {{ menus.build(item.below, depth + 1, attributes) }}
        {% else %}
          <a href="{{ item.url }}" class="{{ depth == 0 ? 'nav-link' : 'dropdown-it
        {% endif %}
      </li>
    {% endfor %}
    </ul>
  {% endif %}
{% endmacro %}
```

Handing the behaviour to Bootstrap deleted 148 lines of hand rolled JavaScript, and brought keyboard navigation, Escape to close, click outside dismissal, and the `aria-expanded` contract with it.

> **Do not add .navbar-nav unless you mean it**
>
> Bootstrap sets `flex-direction: column` on `.navbar-nav` and only flips it to row inside a `.navbar-expand-*` wrapper. Adding that class to a menu list outside a full Bootstrap navbar stacks the menu vertically. Set your own `flex-direction: row` explicitly so no class can flip it.

> **Build it yourself**
>
> What follows is the hands on half: the same ground as above, but written as instructions you can type.

Step 3, The page template (regions & layout)

`templates/page.html.twig` is the shell that wraps every page. It decides which regions are **full-width** (span the browser) vs **boxed** (centred `.container` with side margins), and how sidebars behave.

Create `templates/page.html.twig`:

TWIG

```
<div class="layout-container">

  {# Full-width header #}
  {% if page.header %}
    <header role="banner" class="jarvis-region jarvis-region--full jarvis-header">
      {{ page.header }}
    </header>
  {% endif %}

  {# Boxed nav row: branding left, menu right, responsive offcanvas on mobile #}
  {% if page.primary_menu %}
    <div class="container jarvis-region jarvis-primary-menu d-flex align-items-cent
      <div class="jarvis-branding-wrap">
        {{ page.primary_menu.jarvis_site_branding }}
      </div>
      <button class="jarvis-menu-toggle d-lg-none ms-auto" type="button"
              data-bs-toggle="offcanvas" data-bs-target="#jarvis-primary-nav"
              aria-controls="jarvis-primary-nav" aria-label="{{ 'Toggle main menu'|
        <span class="jarvis-menu-toggle__bar"></span>
      </button>
      <div class="offcanvas-lg offcanvas-end jarvis-primary-nav" tabindex="-1"
           id="jarvis-primary-nav" aria-label="{{ 'Main menu'|t }}">
        <div class="offcanvas-header d-lg-none">
          <button type="button" class="btn-close" data-bs-dismiss="offcanvas"
                  data-bs-target="#jarvis-primary-nav" aria-label="{{ 'Close'|t }}"
        </div>
        <div class="offcanvas-body">
          {{ page.primary_menu.jarvis_main_menu }}
          {{ page.primary_menu|without('jarvis_site_branding', 'jarvis_main_menu')
        </div>
      </div>
    </div>
  {% endif %}
```

```
  {# Boxed breadcrumb #}
  {% if page.breadcrumb %}
    <div class="container jarvis-region jarvis-breadcrumb">{{ page.breadcrumb }}</d
  {% endif %}

  {# Full-width page title — hidden on the front page #}
  {% if page.page_title and not is_front %}
    <div class="jarvis-region jarvis-region--full jarvis-page-title">{{ page.page_t
  {% endif %}

  {% if page.hero %}
    <div class="jarvis-region jarvis-region--full jarvis-hero">{{ page.hero }}</div
  {% endif %}

  {% if page.highlighted %}
    <div class="container jarvis-region jarvis-highlighted">{{ page.highlighted }}<
  {% endif %}

  <main role="main">
    <a id="main-content" tabindex="-1"></a>

    {% if page.content_top %}
      <div class="container jarvis-region jarvis-content-top">{{ page.content_top }
    {% endif %}

    {# A region array is truthy even when empty (it carries cache metadata), so
       render sidebars to a string and test the trimmed output. #}
    {% set sidebar_left = page.sidebar_left|render|trim %}
    {% set sidebar_right = page.sidebar_right|render|trim %}
    {% set has_sidebar = sidebar_left or sidebar_right %}
    {% if has_sidebar %}
      <div class="container jarvis-region jarvis-main">
        <div class="row g-4">
          {% if sidebar_left %}
            <aside class="col-12 col-lg-3 jarvis-sidebar jarvis-sidebar--left" role
              {{ sidebar_left|raw }}
            </aside>
          {% endif %}
          <div class="col jarvis-content">
            {{ page.content }}
          </div>
          {% if sidebar_right %}
            <aside class="col-12 col-lg-3 jarvis-sidebar jarvis-sidebar--right" rol
              {{ sidebar_right|raw }}
            </aside>
```

```
          {% endif %}
        </div>
      </div>
    {% else %}
      {# No sidebars: edge-to-edge so Canvas full-width columns can break out. #}
      <div class="jarvis-content jarvis-content--full">
        {{ page.content }}
      </div>
    {% endif %}

    {% if page.content_bottom %}
      <div class="container jarvis-region jarvis-content-bottom">{{ page.content_bo
    {% endif %}
  </main>

  {% if page.footer %}
    <footer role="contentinfo" class="jarvis-region jarvis-region--full jarvis-foot
      {{ page.footer }}
    </footer>
  {% endif %}

</div>
```

The three ideas a beginner should take away:

1. **`{{ page.REGION }}`** prints a region. The region names match `info.yml` exactly.
2. **Full-width vs boxed** is just: wrap in `.container` (boxed) or don't (full-width).
3. **Two clever bits worth understanding:**
   - `{{ page.primary_menu.jarvis_site_branding }}` renders *one specific block* out of a region by its block ID, so we can split the logo out of the mobile menu drawer. (Those block IDs come from the recipe in Step 11.)
   - A Drupal region is "truthy" even when it holds no blocks (it carries cache metadata), so to know whether a sidebar is *really* empty we render it to a string and `trim` it. When both sidebars are empty the content goes edge-to-edge, which lets a Canvas full-width column layout break out of the centred column.

`ddev drush cr` and reload, you now have a themed page shell.

---

6. The custom modules

jarvis_blocks: rendering content blocks through components

Drupal's custom block types render as a stack of fields. Jarvis already has good looking components, so this module intercepts the render and swaps the field output for the matching component.

`WEB/MODULES/CUSTOM/JARVIS_BLOCKS/SRC/HOOK/JARVISBLOCKSHOOKS.PHP`

```
<?php

declare(strict_types=1);

namespace Drupal\jarvis_blocks\Hook;

use Drupal\Core\Cache\CacheableMetadata;
use Drupal\Core\Hook\Attribute\Hook;
use Drupal\block_content\BlockContentInterface;

final class JarvisBlocksHooks {

  private const MAP = [
    'hero' => 'jarvis:hero',
    'card' => 'jarvis:card',
    'text' => 'jarvis:text',
  ];

  #[Hook('preprocess_block')]
  public function preprocessBlock(array &$variables): void {
    $content = $variables['content'] ?? [];
    $bc = $content['#block_content'] ?? NULL;
    if (!$bc instanceof BlockContentInterface) {
      return;
    }
    $bundle = $bc->bundle();
    if (empty(self::MAP[$bundle])) {
      return;
    }
```

```
    $cacheability = CacheableMetadata::createFromRenderArray($content);
    $props = $this->buildProps($bundle, $bc, $cacheability);

    $component = [
      '#type' => 'component',
      '#component' => self::MAP[$bundle],
      '#props' => $props,
    ];
    // Copy cacheability ONLY, never $content['#cache'] wholesale.
    $cacheability->applyTo($component);
    $variables['content'] = $component;
  }
}
```

Never copy #cache wholesale

That array carries `keys` and `bin`, which name the block's own render cache entry. Reusing them writes your component markup into that entry, so any other render path for the same entity serves it back, or clobbers it. Copy cacheability only, with `CacheableMetadata`.

jarvis_canvas: the Canvas glue

Four hooks, each solving a specific problem.

Hook	Job
`canvas_storable_prop_shape_alter`	Swap Canvas's locked text format for the site's own on opted in props
`node_type_insert`	Create a Canvas template for each new content type
`node_type_delete`	Remove it again, since Drupal will not cascade the delete
`rebuild`	Organise components into folders in the editor

A disabled template is not the same as no template

> Canvas swaps the view display for any content template whose status is TRUE, and unsets `#theme` along with it. Create one enabled but empty, and every node of that type renders as a blank page until somebody opens the editor. Create it disabled instead. It stays editable, it just does not take over output before it has anything to show.

Write hooks as classes

Procedural hook functions are removed in Drupal 12. Use the attribute form in a class under `src/Hook/`. Drupal finds it automatically, and you get constructor injection instead of `\Drupal::` calls.

THE SHAPE

```
namespace Drupal\my_module\Hook;

use Drupal\Core\Hook\Attribute\Hook;
use Drupal\Core\Render\RendererInterface;

final class MyModuleHooks {

  public function __construct(
    private readonly RendererInterface $renderer,
  ) {}

  #[Hook('preprocess_block')]
  public function preprocessBlock(array &$variables): void {
    // ...
  }
}
```

The `.module` file can then be empty, or removed entirely. Drupal 11 does not require one.

> **Build it yourself**
>
> What follows is the hands on half: the same ground as above, but written as instructions you can type.

Step 9, Wiring the theme into Drupal Canvas

So far the components work in any SDC context. To let a site builder drag them onto a page in **Canvas**, two things must be true:

1. **Canvas is installed** (`ddev drush en canvas canvas_field_component -y`).
2. Your theme declares **Canvas page regions**, the areas of the page shell that Canvas is allowed to manage.

A Canvas **page region** is a config entity that mirrors a theme region. Jarvis ships one per region as `canvas.page_region.jarvis.<region>.yml`. Example `canvas.page_region.jarvis.hero.yml`:

YAML

```yaml
langcode: en
status: false
dependencies:
  theme:
    - jarvis
id: jarvis.hero
region: hero
theme: jarvis
component_tree: {  }
```

- **`id: jarvis.hero`** and **`region: hero`** tie this Canvas region to the theme region named `hero` from your `info.yml`.
- **`component_tree: {}`** starts empty, the builder fills it by dragging components in.
- **`status: false`** means the region isn't force-enabled by default; enable the ones you want Canvas to own.

You provide one of these per region you want editable. Once they're imported and cache is cleared, opening Canvas shows your `category: Jarvis` components in the sidebar, and dropping one writes its config into that region's `component_tree`.

> **Where do these files come from?** You don't hand-write the whole `component_tree`, you build a page in the Canvas UI, then export the resulting config with `ddev drush config:export` (or the Canvas export). The YAML you see in

> `recipe/config/` is the *captured result* of arranging components visually, which the recipe replays on a fresh site.

Step 10, Content templates (fielded nodes → components)

A **content template** answers: "when Drupal renders a *Jarvis Sample* node, how should its fields map onto components?" It's a Canvas layout bound to a content type + view mode.

Jarvis ships `canvas.content_template.node.jarvis_sample.full.yml`. The important parts:

YAML

```yaml
id: node.jarvis_sample.full
content_entity_type_id: node
content_entity_type_bundle: jarvis_sample
content_entity_type_view_mode: full
component_tree:
  '0:…':
    component_id: sdc.jarvis.image        # Hero image component
    inputs:
      image:
        sourceType: entity-field
        expression: 'ℹ␜entity:node:jarvis_sample␝field_hero_banner␞…'   # ← binds a
      image_style: hero_banner
      full_width: true
  '1:…':
    component_id: sdc.jarvis.section
    inputs: { width: normal, bg: none }
  '1:content:0:…':
    parent_uuid: …                        # nested inside the section's slot
    slot: content
    component_id: field_display.field_display
    inputs:
      field_name: field_publication_date
      formatter_id: datetime_default
  '2:first:0:…':
    component_id: sdc.jarvis.text
```

```
    inputs:
      body:
        sourceType: entity-field
        expression: 'ℹ␜entity:node:jarvis_sample␝field_body␟processed'  # body fie
```

The concepts a beginner needs:

- `component_id: sdc.jarvis.image`, references your SDC by machine path (`sdc.<theme>.<component>`).
- `inputs` are the component's props. A static value (`image_style: hero_banner`) is hard-coded; a `sourceType: entity-field` input with an `expression` *binds a node field* to the prop. That funky `ℹ␜entity:node…␝field_body␟` string is Canvas's field-path syntax, you don't type it by hand, Canvas writes it when you click "link to field" (the 🔗 icon) in the UI.
- **Nesting** (`parent_uuid` + `slot`) is how a component lands *inside another component's slot*, here, a field display sits inside the Section's `content` slot.
- **Remember the binding rule:** a required prop only exposes the 🔗 link icon when the field it binds to is also required. And for image styles inside a template, use the SDC's own `|image_style` (via `twig_tweak`), content templates can't use Canvas adapters.

To *style a whole content type's fields*, you edit its content template (a ContentTemplate config entity), you don't add a "styling field" to the node.

---

7. Build order: making one yourself

Starting from nothing, this order avoids the most rework.

1. **Skeleton.** Create `jarvis.info.yml` with a name, type, core version, and regions. Add `jarvis.libraries.yml` with one global library. Install the theme and confirm it

renders.

2. **Page template.** Override `page.html.twig` and print each region. Decide which are full width and which are boxed. Get the shell right before styling anything.
3. **Base CSS.** Bring in Bootstrap. Add `overrides.css` for what Bootstrap does not cover.
4. **First component.** Build the simplest one, something like Text. Two files. Render it from a Twig template to prove the pipeline works.
5. **Shared helpers.** Once three or four components repeat the same props, extract the macro. Do not do this first. You will not know what to share until you have felt the repetition.
6. **The rest of the components.** Now they go quickly, because the patterns are set.
7. **Theme settings.** Add `theme-settings.php` and the preprocess hook that turns settings into CSS variables.
8. **JavaScript.** Add behaviors for anything that needs measuring or interaction. Keep the maths in one file.
9. **Modules.** Only when you hit something a theme cannot do. Not before.
10. **Recipe.** Export configuration and demo content so the whole thing installs from empty.

> **Test the install from empty, early and often**
>
> A recipe that has never been installed on a blank database is a recipe that does not work. On this project the first real attempt failed four separate times, each a different collision between shipped configuration and configuration Drupal creates on its own. Every one of them was invisible on the development site.

> **Build it yourself**
>
> What follows is the hands on half: the same ground as above, but written as instructions you can type.

Step 11, Package it all as a recipe

A **recipe** installs the theme + its dependencies + config + demo content in one command. It lives in `recipe/` and is driven by `recipe/recipe.yml`.

YAML

```yaml
name: 'Jarvis Theme'
description: 'Enables the Jarvis theme, required contrib modules, base config, the
  Sample content type, and demo content. Requires Canvas + the Jarvis theme install
  cache rebuilt first (see README).'
type: 'Theme'
install:
  - twig_tweak
  - focal_point
  - canvas
  - canvas_field_component
  - jarvis
  - media
  - media_library
  - image
  - menu_ui
  - menu_link_content
  - datetime
  - options
  - path
config:
  actions:
    system.theme:
      simpleConfigUpdate:
        default: jarvis
    block.block.jarvis_main_menu:
      simpleConfigUpdate:
        status: true
        region: primary_menu
        settings:
          id: 'system_menu_block:main'
          level: 1
          depth: 2
          expand_all_items: true
    block.block.jarvis_site_branding:
      simpleConfigUpdate:
        status: true
        region: primary_menu
        settings:
          id: system_branding_block
          use_site_logo: true
          use_site_name: false
    # …more block placements (footer menu, breadcrumbs, messages, page title, …)
```

The three sections:

1. `install:`, every module (and the theme itself) to turn on, in dependency order. This is why applying the recipe on a clean site "just works."
2. `config.actions:`, surgical config changes. `simpleConfigUpdate` sets values (e.g. make `jarvis` the default theme) and **places blocks into regions**, this is what populates your `primary_menu`, `footer`, etc. Note the block IDs like `jarvis_site_branding` and `jarvis_main_menu`, *these are the exact IDs the page template renders by name* in Step 3.
3. **Config & content files** (in `recipe/config/` and `recipe/content/`), everything else is imported wholesale:
   - `recipe/config/` holds the field storage/instances, the `jarvis_sample` node type, image styles (`hero_banner`, `wide`), the focal-point crop type, media types, the Canvas page regions (Step 9), the content template (Step 10), and `jarvis.settings.yml` (the default colours/fonts).
   - `recipe/content/` holds demo content as YAML (a Test Blog node, a Test Page Canvas page, media, files, menu links) via **default_content**. UUID-named files are entities; the raw `.png` / `.jpg` are the referenced image files.

To build these config files, you *don't* write them by hand. You configure everything on a working site (create the content type, add fields, set up Canvas, pick colours), then export:

BASH

```
ddev drush config:export
# copy the relevant jarvis.*, canvas.*, field.*, node.type.*, image.style.* files
# into web/themes/custom/jarvis/recipe/config/
```

`recipe/config/jarvis.settings.yml` is worth a peek, it's the exported default theme settings, so a fresh install starts with the right brand colours and fonts already chosen:

YAML

```
logo:
  use_default: 1
jarvis_color_primary: '#2d6cdf'
jarvis_color_secondary: '#1b2a4a'
jarvis_color_title_bg: '#2d6cdf'
jarvis_color_footer_bg: '#000000'
jarvis_font_base_family: 'Open Sans'
```

```
jarvis_font_base_weight: '300'
jarvis_font_headings_family: 'Open Sans'
jarvis_font_headings_weight: '600'
# …etc…
```

Step 12, Install, apply, and test

Because the recipe's Canvas config references the theme, the **order matters**. On a fresh site:

BASH

```
# Apply the recipe against a freshly installed site. It installs Canvas,
# canvas_field_component and the theme itself, in the right order, on its own.
ddev drush recipe /var/www/html/recipes/jarvis

# Rebuild caches and log in.
ddev drush cr
ddev drush uli
```

> **Do not pre-install Canvas or the theme.** Running `theme:enable jarvis` plus `en canvas` plus `cr` before the recipe breaks it. That rebuild makes Canvas create its component entities and makes Drupal place the theme's blocks, and a recipe refuses to import config that already exists and differs. You get `The configuration '...' exists already and does not match the recipe's configuration`. The recipe lists both modules and the theme in its own `install:` list.

Then verify, in order:

1. **Front end loads** with Bootstrap styling and your logo in the nav. → theme + libraries OK.
2. **`/admin/appearance/settings/jarvis`** shows the Colours and Fonts sections. Change a colour, save, reload → the site re-skins. → Steps 7-8 OK.

3. **Save the Fonts form** → check `sites/default/files/jarvis-fonts/jarvis-fonts.css` exists and the site uses the new fonts. → self-hosting OK.
4. **Edit the Test Page in Canvas** → your `Jarvis` and `Jarvis Layout` components appear in the sidebar; drag a Hero in, set its image, save, view the page. → Steps 4-6, 9 OK.
5. **View the Test Blog node** → its hero field renders as the Hero component, body as Text, per the content template. → Step 10 OK.

If something's missing, it's almost always caching or order, see below.

---

8. Hard won lessons

These cost real debugging time. They are here so they cost you less.

Verify by rendering, not by reading

Checking that a file contains the right text is not verification. Twice on this project a change looked correct in the source, was served correctly, and still broke the page: once a Twig comment inside a tag, once a Bootstrap class that flipped a flex direction. Render the thing and check the result. In PHP use `renderInIsolation()`. In the browser use `getComputedStyle`.

A text format nobody can use is not installed

Shipping `filter.format.my_format` is half the job. Text formats are permission gated, so without a matching grant only user 1 can use them. User 1 bypasses every permission check, so everything looks fine to you while editors see nothing. Grant the permission explicitly, and grant a restricted format rather than an unrestricted one.

Recipes refuse configuration that already exists and differs

If installing a theme or module makes Drupal auto create something your recipe also ships, the install aborts. Either ship values that match what gets auto created and change them afterwards with a config action, or do not ship that configuration at all.

Config actions run after config import

So an action cannot create something that another shipped configuration validates against during import. If configuration A references configuration B, then B has to be a real file.

Patch files need wiring, not just existence

A patch sitting in a `patches/` folder does nothing on its own. Without a `composer-patches` entry it never re applies, and if the module directory is gitignored the fix exists only on the machine where it was made. Add `composer-exit-on-patch-failure` so a stale patch fails loudly instead of silently shipping unpatched code.

Prefer deleting to adding

The largest improvements on this project were deletions. Handing the menu to Bootstrap removed 148 lines of JavaScript and about 60 of CSS, and improved accessibility at the same time. Collapsing the colour blocks removed 35 lines. Sharing one contrast engine removed roughly 130. Less code, fewer places for a bug to hide.

Jarvis is a component driven Drupal 11 theme built on Bootstrap 5, for Drupal Canvas and Layout Builder, with WCAG 2.2 AA safeguards built in.

Written for DrupalHelps. Every code sample here is taken from the working theme.

> **Build it yourself**

> What follows is the hands on half: the same ground as above, but written as instructions you can type.

Troubleshooting & cheat sheet

"My new component / region / setting doesn't appear." → `ddev drush cr`. SDC, libraries, regions, and theme settings are all discovery-cached.

"The recipe failed on a Canvas config import." → You applied it before Canvas or the theme was installed. Do Step 12 in order: theme → Canvas → `cr` → recipe.

"A component prop won't bind to a field (no 🔗 icon in Canvas)." → The SDC prop is `required` but the target content field is not required. Make the field required, or drop `required` from the prop.

"The Canvas preview looks broken / clipped." → You used `vh` / `vw` somewhere. Canvas renders in an iframe with no real viewport. Use `rem` / `px` / `%`.

"I need a named image style in a component." → Pass the raw file URI as a prop and use `twig_tweak`'s `|image_style('hero_banner')` in the SDC. Don't use a Canvas adapter inside a content template.

"How do I regenerate the recipe after changing things in the UI?" → `ddev drush config:export`, then copy the changed `jarvis.* / canvas.* / field.* / node.type.* / image.style.*` files into `recipe/config/`.

The seven files that define the theme

File	Job
`jarvis.info.yml`	Declares the theme + its regions.
`jarvis.libraries.yml`	Declares CSS/JS bundles (Bootstrap, settings assets).
`templates/page.html.twig`	The page shell / region layout.
`css/overrides.css`	Bootstrap-variable theming + `jp-` spacing utilities.
`jarvis.theme`	Preprocess hooks: output colours + attach fonts.

File	Job
`theme-settings.php`	The colour + font admin form and its handlers.
`components/*/`	The SDC kit, one folder per component.

The one pattern that repeats everywhere

Every Jarvis component is: **`.component.yml` (props via JSON-Schema + `enum` / `meta:enum` dropdowns, `$ref` for images, `slots` for nesting) → `.twig` (map props onto Bootstrap classes, reuse the `jp-` spacing snippet)**. Learn it once on the Text component, and Hero, Card, Image, Video, and the column layouts are all variations on it.

Happy theming.

---

9. Every component, one at a time

Nineteen components ship with the theme. Each entry says what the component is for, lists every prop and slot, and walks through placing one. If you have never touched a Single Directory Component before, start with Section and Text, then work down the list.

> **How to read these tables**
>
> A **prop** is a setting on the component. It appears as a field in the Canvas settings tray, labelled with the text in the Label column. **Required** means Canvas will not let you save without a value. A prop with a list of choices renders as a dropdown. A **slot** is a different thing: a hole you drag other components into.

Layout components

Wrappers you drop other components into. Nearly every page starts with one of these.

Section `jarvis:section`

Container/section wrapper that holds other components in its slot.

Use it when: You want a band of colour, or a contained width, around whatever you drop inside.

Section is the simplest wrapper in the theme. One slot, a small set of props. Use it when you want a background colour behind a run of content, or when you want to control whether that content is contained or edge to edge.

The background choices map onto the theme's colour contract, so whatever you drop inside inherits readable text automatically. Choose Dark, and the headings, body text, and links inside all flip to white without you touching them.

> **Placing a Section in Canvas**
>
> 1. Open the page in Canvas and click the plus button.
> 2. Choose Section from the Sections folder.
> 3. In the settings tray, set Background to Dark.
> 4. Drag a Text component into the Content slot.
> 5. The text turns white on its own. That is the colour contract doing its job.

Props

Prop	Label	Type	Required	Choices and default
`width`	Width	`string`	No	`normal`, `full` Default `normal`
`bg`	Background	`string`	No	`none`, `light`, `dark`, `primary`, `body-tertiary` Default `none`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`,

Prop	Label	Type	Required	Choices and default
				`right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`
`no_padding`	Remove padding (let content touch)	`boolean`	No	Default `False`

Slots you can drag components into:

Slot	Label in Canvas
`content`	Content

CALLING IT FROM A TWIG TEMPLATE

```
{% embed 'jarvis:section' with {
  width: 'normal',
  bg: 'none'
} %}
  {% block content %}Your content{% endblock %}
{% endembed %}
```

> **Worth knowing**

> Section adds no padding of its own beyond the vertical padding toggle. If your content is touching the edges, set Padding to All rather than reaching for custom CSS.

1 Column `jarvis:one-column`

Single-column row. Content spans the full width.

Use it when: You want one full width band with a background image and an overlay.

One Column looks redundant next to Section until you need a background image. Section handles solid colours. The column layouts add image backgrounds, an adjustable dark or light overlay, and the automatic contrast tuning that goes with them.

Set Width to Full and the inner container goes edge to edge. Leave it Boxed and the content stays inside the site's normal reading width while the background still spans the viewport.

> **A photo band with readable text**
>
> 1. Add One Column.
> 2. Set Background to Image and pick a photo in Background image.
> 3. Set Width to Full.
> 4. Drop a Text component into the first slot.
> 5. Watch the overlay badge. If it shows a cross, raise Overlay until it shows a tick.

Props

Prop	Label	Type	Required	Choices and default
`width`	Section width	`string`	No	`boxed`, `full` Default `boxed`
`background`	Background colour	`string`	No	`none`, `primary`, `secondary`, `bg1`, `bg2`, `dark`, `light`, `image` Default `none`

Prop	Label	Type	Required	Choices and default
`background_image`	Background image	`image`	No	Pick a media item
`overlay`	Image overlay opacity (%)	`integer`	No	Default `45`
`text_color`	Text colour (overrides child components)	`string`	No	`inherit`, `primary`, `secondary`, `black`, `white` Default `inherit`
`gap`	Column gap	`string`	No	`none`, `small`, `normal`, `large` Default `normal`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

Slots you can drag components into:

Slot	Label in Canvas
`first`	Column

CALLING IT FROM A TWIG TEMPLATE

```
{% embed 'jarvis:one-column' with {
  width: 'boxed',
  background: 'none'
} %}
  {% block first %}Your content{% endblock %}
{% endembed %}
```

> **Worth knowing**
>
> Overlay is a floor, not a fixed value. Set 45 and, if the photo is bright, the live site raises it until white text clears WCAG AA. The editor badge shows what the number needs to be, so the design you approve is the design that ships.

2 Columns `jarvis:two-column`

Two columns side by side on desktop, stacked full-width on mobile.

Use it when: Two things side by side on desktop that should stack on a phone.

Two Column gives you two slots and a Ratio prop. Even splits down the middle. Wide left gives the first slot two thirds. Wide right does the opposite.

Both columns stack to full width below the medium breakpoint, so there is no separate mobile arrangement to manage. Everything else, the backgrounds, the overlay, the contrast handling, works as it does in One Column, because all three column layouts share one shell template.

> **Text beside an image**
>
> 1. Add Two Column and set Ratio to Wide left.
> 2. Drop a Text component into Column 1.
> 3. Drop an Image component into Column 2.
> 4. Set Gap to Large if the two feel cramped.

Props

Prop	Label	Type	Required	Choices and default
`width`	Section width	`string`	No	`boxed`, `full` Default `boxed`
`background`	Background colour	`string`	No	`none`, `primary`, `secondary`, `bg1`, `bg2`, `dark`, `light`, `image` Default `none`
`background_image`	Background image	`image`	No	Pick a media item
`overlay`	Image overlay opacity (%)	`integer`	No	Default `45`
`text_color`	Text colour (overrides child components)	`string`	No	`inherit`, `primary`, `secondary`, `black`, `white` Default `inherit`
`gap`	Column gap	`string`	No	`none`, `small`, `normal`, `large` Default `normal`
`ratio`	Column ratio	`string`	No	`even`, `wide-left`, `wide-right` Default `even`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`

Prop	Label	Type	Required	Choices and default
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

Slots you can drag components into:

Slot	Label in Canvas
`first`	Column 1
`second`	Column 2

CALLING IT FROM A TWIG TEMPLATE

```
{% embed 'jarvis:two-column' with {
  width: 'boxed',
  background: 'none'
} %}
  {% block first %}Your content{% endblock %}
  {% block second %}Your content{% endblock %}
{% endembed %}
```

> **Worth knowing**
>
> Gap controls space between the columns, not around them. For space around the whole band, use Padding.

3 Columns `jarvis:three-column`

Three columns on desktop, two on tablet, stacked full-width on mobile.

Use it when: Three equal items, usually cards, stats, or people.

Three Column stacks to one column on phones, goes two across on tablets, then three across on desktop. That middle step matters. Three narrow columns on a tablet become

unreadable, so the layout drops to two first.

It is the natural home for a row of Cards, Stats, or People.

> **A row of three stats**
>
> 1. Add Three Column.
> 2. Drop a Stat component into each of the three slots.
> 3. Set Background to Primary on the Three Column.
> 4. The stat numbers stay readable, because the theme swaps their colour for you.

Props

Prop	Label	Type	Required	Choices and default
`width`	Section width	`string`	No	`boxed`, `full` Default `boxed`
`background`	Background colour	`string`	No	`none`, `primary`, `secondary`, `bg1`, `bg2`, `dark`, `light`, `image` Default `none`
`background_image`	Background image	`image`	No	Pick a media item
`overlay`	Image overlay opacity (%)	`integer`	No	Default `45`
`text_color`	Text colour (overrides child components)	`string`	No	`inherit`, `primary`, `secondary`, `black`, `white` Default `inherit`
`gap`	Column gap	`string`	No	`none`, `small`, `normal`, `large` Default `normal`

Prop	Label	Type	Required	Choices and default
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

Slots you can drag components into:

Slot	Label in Canvas
`first`	Column 1
`second`	Column 2
`third`	Column 3

CALLING IT FROM A TWIG TEMPLATE

```
{% embed 'jarvis:three-column' with {
  width: 'boxed',
  background: 'none'
} %}
  {% block first %}Your content{% endblock %}
  {% block second %}Your content{% endblock %}
  {% block third %}Your content{% endblock %}
{% endembed %}
```

> **Worth knowing**
>
> Put Cards inside a coloured Three Column and the cards keep their own light background and dark text. That is deliberate. White card text on a white card would vanish, so the theme resets the colour contract inside cards.

Content components

The things a visitor actually reads.

Text `jarvis:text`

Rich text block with width and alignment controls.

Use it when: A run of prose, with control over reading width and alignment.

Text is the component you will use most. It gives you a rich text field plus two things a plain body field does not: a Width prop that constrains line length, and an Alignment prop.

Reading width matters more than people expect. Long lines are genuinely harder to read. Narrow gives roughly 50 to 60 characters per line, Normal about 70, and Wide fills the container.

> **A centred introduction**
>
> 1. Add Text.
> 2. Type your paragraph in the Body field.
> 3. Set Width to Narrow and Alignment to Center.
> 4. That combination is the standard section introduction in this theme.

Props

Prop	Label	Type	Required	Choices and default
`body`	Body	`string`	Yes	Free text

Prop	Label	Type	Required	Choices and default
`width`	Width	`string`	No	`narrow`, `normal`, `wide` Default `normal`
`align`	Alignment	`string`	No	`start`, `center`, `end` Default `start`
`color`	Text colour	`string`	No	`inherit`, `primary`, `secondary`, `black`, `white` Default `inherit`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:text', {
  body: 'Body'
}) }}
```

> **Worth knowing**

> Use Text for prose. Use WYSIWYG when you need the full toolbar with media embeds and tables. Text is the lighter of the two.

WYSIWYG `jarvis:wysiwyg`

Freeform CKEditor rich-text block. Full-width by default, with an optional readable measure.

Use it when: Freeform content with the full editor: media, tables, lists, links.

WYSIWYG is the escape hatch. Where Text is deliberately constrained, this hands the editor the whole CKEditor toolbar, including embedded media and tables.

It is also the only component wired to the site's own text formats rather than Canvas's locked one. That is what the `x-jarvis-html-format` marker in its schema does.

> **A rich article body**
>
> 1. Add WYSIWYG.
> 2. Click into the Body field and the full toolbar appears.
> 3. Use the media button to embed an image from the media library.
> 4. The editing area uses the theme's typography, so what you see matches what publishes.

Props

Prop	Label	Type	Required	Choices and default
`body`	Body	`string`	Yes	Free text
`measure`	Text measure	`string`	No	`full`, `readable` Default `full`
`align`	Alignment	`string`	No	`start`, `center`, `end` Default `start`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`

Prop	Label	Type	Required	Choices and default
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:wysiwyg', {
  body: 'Body'
}) }}
```

> **Worth knowing**
>
> If you find yourself pasting raw HTML into a Text component, switch to WYSIWYG. That is what it is for.

Hero `jarvis:hero`

Full-width hero banner with heading, subtext, CTA and background image.

Use it when: The banner at the top of a landing page.

Hero is a full width banner: a heading, optional subheading, an optional call to action button, and usually a background image with a darkening overlay.

Heading level defaults to `h1`, because a hero is normally the page's main heading. Place two heroes on one page and change the second to `h2`. A page should have exactly one `h1`.

A landing page hero

1. Add Hero at the top of the page.
2. Set Heading and leave Heading level at h1.
3. Pick a Background image.
4. Set CTA text to "Get started" and CTA link to your target page.
5. Check the overlay badge shows a tick before you publish.

Props

Prop	Label	Type	Required	Choices and default
`heading`	Heading	`string`	Yes	Free text
`heading_level`	Heading level	`string`	No	`h1`, `h2`, `h3` Default `h1`
`subheading`	Subheading	`string`	No	Free text
`cta_text`	CTA text	`string`	No	Free text
`cta_href`	CTA link	`string`	No	Free text
`image`	Background image	`image`	No	Pick a media item
`overlay`	Overlay opacity (%)	`integer`	No	Default `40`
`align`	Text alignment	`string`	No	`start`, `center`, `end` Default `center`
`text_color`	Text color	`string`	No	`light`, `dark` Default `light`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`

Prop	Label	Type	Required	Choices and default
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:hero', {
  heading: 'Heading'
}) }}
```

> **Worth knowing**
>
> Text colour Dark tells the theme the text sits on the bare image, so it skips the overlay entirely. Use it only on genuinely light photos, because nothing auto corrects for you in that mode.

Card `jarvis:card`

Flexible card with image at top, left, right, or as background.

Use it when: A self contained unit: image, title, short text, and a link.

Card has four layouts, chosen with the Variant prop. Image top is the classic stacked card. Image left and Image right put the picture beside the text. Background puts the image behind everything, which is where the overlay and contrast handling come in.

Heading level defaults to `h3`, usually right for a card sitting under a section heading. Change it if your page outline needs something else.

> **Three linked cards**
>
> 1. Add a Three Column layout.
> 2. Drop a Card into each slot.
> 3. On each card set Variant to Image top, pick an image, set Title and Body.
> 4. Set Link text and Link so the card leads somewhere.

Props

Prop	Label	Type	Required	Choices and default
`variant`	Image position	`string`	No	`image-top`, `image-left`, `image-right`, `background` Default `image-top`
`overlay`	Overlay opacity (%)	`integer`	No	Default `40`
`text_color`	Text color	`string`	No	`light`, `black` Default `light`
`image`	Image	`image`	No	Pick a media item
`title`	Title	`string`	Yes	Free text
`heading_level`	Heading level	`string`	No	`h2`, `h3`, `h4` Default `h3`
`body`	Body	`string`	No	Free text
`link_text`	Link text	`string`	No	Free text
`link_href`	Link URL	`string`	No	Free text
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`

Prop	Label	Type	Required	Choices and default
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:card', {
  title: 'Title'
}) }}
```

> **Worth knowing**
>
> Background is the only variant that uses Overlay and Text colour. In the other three the image sits beside or above the text, so there is nothing to darken.

Card Full Image `jarvis:card-full-image`

Full-width 2-column band: an image that fills its half edge-to-edge beside a rich-text column.

Use it when: A two up band where the picture bleeds to the edge of its half.

Card Full Image splits into two halves. One half is media, the other content, and the image fills its half completely rather than sitting inside padding.

Supply the image with the Image prop, or drop an Image component into the Image slot if you need image style control. The prop wins when both are set.

> **An alternating feature row**
>
> 1. Add Card Full Image.
> 2. Set the Image prop.
> 3. Drop a Text or WYSIWYG component into the Content slot.
> 4. Add a second below and tick Image on right so the two mirror each other.

Props

Prop	Label	Type	Required	Choices and default
`image`	Image	`image`	No	Pick a media item
`image_right`	Image on right	`boolean`	No	Default `False`
`fade_edge`	Fade image edge next to text	`boolean`	No	Default `False`
`background`	Background colour	`string`	No	`none`, `primary`, `secondary`, `bg1`, `bg2`, `dark`, `light` Default `none`

Slots you can drag components into:

Slot	Label in Canvas
`media`	Image
`content`	Content

CALLING IT FROM A TWIG TEMPLATE

```
{% embed 'jarvis:card-full-image' with {
  image: media_image,
  image_right: false
} %}
  {% block media %}Your content{% endblock %}
```

```
  {% block content %}Your content{% endblock %}
{% endembed %}
```

> **Worth knowing**
>
> Image on right flips the arrangement. Alternate it down a page so a stack of these does not look monotonous.

Call to Action `jarvis:cta`

Centered call-to-action: overline subtitle, heading, text and a single button set by the Button text / Button link props.

Use it when: A short, centred prompt to do one thing.

CTA is a centred block: an optional overline, a heading, a line or two of text, and one button. Deliberately narrow in scope. One message, one action.

The button colour adapts to the background. On a coloured band it goes light, on a plain background it uses the site's primary colour.

> **A newsletter prompt**
>
> 1. Add CTA inside a Section with Background set to Primary.
> 2. Set Heading to your prompt.
> 3. Set Button text and Button link.
> 4. The button turns light automatically against the coloured band.

Props

Prop	Label	Type	Required	Choices and default
`subtitle`	Subtitle	`string`	No	Free text
`heading`	Heading	`string`	Yes	Free text

Prop	Label	Type	Required	Choices and default
`heading_level`	Heading level	`string`	No	`h2`, `h3`, `h4` Default `h2`
`text`	Text	`string`	No	Free text
`button_text`	Button text	`string`	No	Free text
`button_href`	Button link	`string`	No	Free text
`bg`	Background color	`string`	No	`none`, `primary`, `secondary` Default `none`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:cta', {
  heading: 'Heading'
}) }}
```

> **Worth knowing**

> Need more than one button, or a longer block of supporting text? Use Large Call To Action instead. CTA is the compact version.

Large Call to Action `jarvis:large-call-to-action`

Eyebrow, large heading, lead text, three icon+sentence rows separated by rules, a primary button and a reversed-colour link.

Use it when: A full width feature block with more room than CTA.

Large Call To Action is the heavyweight version: more props, an optional icon, room for longer supporting copy, and more layout control.

Reach for it when a CTA feels cramped, typically at the foot of a long landing page where the closing message should carry some weight.

> **A closing block**
>
> 1. Add Large Call To Action at the foot of the page.
> 2. Set Heading and the supporting text.
> 3. Choose an icon if the design calls for one.
> 4. Set the button text and link.

Props

Prop	Label	Type	Required	Choices and default
`eyebrow`	Eyebrow	`string`	No	Free text
`heading`	Heading	`string`	Yes	Free text
`heading_level`	Heading level	`string`	No	`h2`, `h3`, `h4` Default `h2`
`text`	Lead text	`string`	No	Free text
`row1_icon`	Row 1 icon	`string`	No	`none`, `fa-solid fa-check`, `fa-solid fa-circle-check`, `fa-`

Prop	Label	Type	Required	Choices and default
				`solid fa-cross`, `fa-solid fa-book-bible`, `fa-solid fa-church`, `fa-solid fa-hands-praying`, `fa-solid fa-dove`, `fa-solid fa-heart`, `fa-solid fa-people-group`, `fa-solid fa-star`, `fa-solid fa-arrow-right` Default `fa-solid fa-check`
`row1_text`	Row 1 text	`string`	No	Free text
`row2_icon`	Row 2 icon	`string`	No	`none`, `fa-solid fa-check`, `fa-solid fa-circle-check`, `fa-solid fa-cross`, `fa-solid fa-book-bible`, `fa-solid fa-church`, `fa-solid fa-hands-praying`, `fa-solid fa-dove`, `fa-solid fa-heart`, `fa-solid fa-people-group`, `fa-solid fa-star`, `fa-solid fa-arrow-right` Default `fa-solid fa-check`
`row2_text`	Row 2 text	`string`	No	Free text

Prop	Label	Type	Required	Choices and default
`row3_icon`	Row 3 icon	`string`	No	`none`, `fa-solid fa-check`, `fa-solid fa-circle-check`, `fa-solid fa-cross`, `fa-solid fa-book-bible`, `fa-solid fa-church`, `fa-solid fa-hands-praying`, `fa-solid fa-dove`, `fa-solid fa-heart`, `fa-solid fa-people-group`, `fa-solid fa-star`, `fa-solid fa-arrow-right` Default `fa-solid fa-check`
`row3_text`	Row 3 text	`string`	No	Free text
`button_text`	Button text	`string`	No	Free text
`button_href`	Button link	`string`	No	Free text
`link_text`	Link text	`string`	No	Free text
`link_href`	Link URL	`string`	No	Free text
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`

Prop	Label	Type	Required	Choices and default
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:large-call-to-action', {
  heading: 'Heading'
}) }}
```

> **Worth knowing**
>
> It carries a lot of props. Set Heading first, preview, then adjust. Changing everything before you look makes it hard to tell which prop did what.

Stat card `jarvis:stat`

Centered stat: icon, big number, description.

Use it when: A single number worth drawing attention to.

Stat is a centred icon, a large number, and a short description. Title holds the number, which reads oddly in the form but keeps the schema simple.

Count up animates the number when it scrolls into view. It respects the operating system's reduced motion setting, so it will not animate for people who asked it not to.

> **A row of results**
>
> 1. Add a Three Column layout.
> 2. Drop a Stat into each slot.
> 3. Set Title to the number, for example 1,200+.
> 4. Put the label in Body, for example "Happy customers served".

> 5. Tick Count up if you want the animation.

Props

Prop	Label	Type	Required	Choices and default
`icon`	Icon	`image`	No	Pick a media item
`title`	Number	`string`	Yes	Free text
`number_color`	Number color	`string`	No	`primary`, `secondary`, `dark`, `light` Default `primary`
`count_up`	Count-up animation	`boolean`	No	Default `False`
`body`	Description	`string`	No	Free text
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:stat', {
  title: 'Number'
```

```
}) }}
```

> **Worth knowing**
>
> Number colour is checked against the surrounding background. Place a Stat on a dark band and the theme swaps the number to something readable rather than leaving it in a colour that disappears.

Person `jarvis:person`

Portrait, name, position, phone, email, and a link.

Use it when: A team member, speaker, or author.

Person carries a portrait, a name, a position, contact details, and up to six social links. The Variant prop chooses between a bordered card, a plain arrangement, and an image beside the text.

The name renders as a real heading element, chosen with Heading level. It looks like a heading, so it is one. Somebody navigating by heading will find it.

> **A team row**
>
> 1. Add a Three Column layout.
> 2. Drop a Person into each slot.
> 3. Set Variant to Card, add the portrait, Name, and Position.
> 4. Add LinkedIn and any other social URLs you want shown.

Props

Prop	Label	Type	Required	Choices and default
`variant`	Layout	`string`	No	`card`, `plain`, `image-left` Default `card`

Prop	Label	Type	Required	Choices and default
`image`	Photo	`image`	No	Pick a media item
`image_uri`	Image source URI (internal)	`string`	No	Free text
`heading_level`	Name heading level	`string`	No	`h2`, `h3`, `h4`, `h5` Default `h3`
`name`	Name	`string`	No	Default `Jane Doe`
`position`	Position	`string`	No	Free text
`phone`	Phone	`string`	No	Free text
`email`	Email	`string`	No	Free text
`link_text`	Link text	`string`	No	Free text
`link_href`	Link URL	`string`	No	Free text
`linkedin`	LinkedIn URL	`string`	No	Free text
`x`	X (Twitter) URL	`string`	No	Free text
`facebook`	Facebook URL	`string`	No	Free text
`instagram`	Instagram URL	`string`	No	Free text
`youtube`	YouTube URL	`string`	No	Free text
`github`	GitHub URL	`string`	No	Free text
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`

Prop	Label	Type	Required	Choices and default
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:person', {
  variant: 'card',
  image: media_image
}) }}
```

> **Worth knowing**
>
> The portrait is cropped server side to the Portrait image style, so upload something reasonably large and let the theme handle the crop.

Button `jarvis:button`

A single call-to-action button (Bootstrap styles).

Use it when: A standalone link styled as a button.

Button is the smallest component in the set: text, a link, a colour variant, a size, and an option to open in a new tab.

Most of the time you will not place one directly. Hero, CTA, Card, and the rest carry their own button props. Use this when you need a button on its own, outside any of those.

> **A button under some text**
>
> 1. Add Text with your paragraph.
> 2. Add Button below it.
> 3. Set Text and Link.

4. Choose a Variant, for example Outline primary, and a Size.

Props

Prop	Label	Type	Required	Choices and default
`text`	Text	`string`	Yes	Default `Get started`
`href`	Link URL	`string`	No	Free text
`variant`	Style	`string`	No	`primary`, `secondary`, `outline-primary`, `outline-secondary`, `light`, `dark` Default `primary`
`size`	Size	`string`	No	`small`, `medium`, `large` Default `medium`
`new_tab`	Open in new tab	`boolean`	No	Default `False`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:button', {
  text: 'Text'
}) }}
```

Worth knowing

Ticking "open in new tab" adds a visually hidden "(opens in new tab)" note for screen reader users, plus the security attributes browsers want. Do not hand roll that.

Media components

Pictures and video, with the accessibility handling built in.

Image `jarvis:image`

Responsive image with optional caption, full-width option.

Use it when: A single picture, optionally with a caption.

Image renders a picture with optional caption, rounded corners, and a choice of image style. Full width takes it edge to edge, otherwise it stays inside the container.

Priority is worth understanding. Tick it for an image visible without scrolling and the browser loads it eagerly at high priority. Leave it unticked for everything else so the image loads lazily.

> **A captioned figure**
>
> 1. Add Image.
> 2. Pick a media item.
> 3. Type a Caption.
> 4. Choose an Image style, for example Wide, so the file served matches the space it fills.

Props

Prop	Label	Type	Required	Choices and default
`image`	Image	`image`	No	Pick a media item
`image_style`	Image style	`string`	No	`original`, `thumbnail`, `medium`, `large`, `wide`, `hero_banner` Default `original`
`image_uri`	Image source URI (internal)	`string`	No	Free text
`caption`	Caption	`string`	No	Free text
`full_width`	Full width	`boolean`	No	Default `False`
`match_height`	Match column height	`boolean`	No	Default `False`

Prop	Label	Type	Required	Choices and default
`rounded`	Rounded corners	`boolean`	No	Default `True`
`priority`	Load eagerly (above the fold)	`boolean`	No	Default `False`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:image', {
  image: media_image,
  image_style: 'original'
}) }}
```

> **Worth knowing**
>
> Alt text comes from the media item, not the component. Set it when you upload, and set it empty deliberately if the image is purely decorative.

Image Overlay `jarvis:image-overlay`

An image with a coloured text card overlaid in the lower corner (left or right).

Use it when: A picture with a caption card sitting over one corner.

Image Overlay puts a small panel of text on top of the picture, anchored to a corner you choose with the Position prop. The panel has its own background, so the text stays readable regardless of what is behind it.

Overlay opacity controls how strongly that panel is tinted.

> **A captioned hero image**
>
> 1. Add Image Overlay.
> 2. Pick the image.
> 3. Set Heading and Text.
> 4. Set Position to whichever corner keeps the panel clear of the subject of the photo.

Props

Prop	Label	Type	Required	Choices and default
`image`	Image	`image`	No	Pick a media item
`heading_level`	Heading level	`string`	No	`h2`, `h3`, `h4`, `h5` Default `h3`
`heading`	Overlay heading	`string`	No	Free text
`text`	Overlay text	`string`	No	Free text
`position`	Overlay position	`string`	No	`left`, `right` Default `left`
`background`	Overlay colour	`string`	No	`primary`, `secondary`, `bg1`, `bg2`, `dark`, `light`

Prop	Label	Type	Required	Choices and default
				Default `primary`
`overlay_opacity`	Overlay opacity (%)	`integer`	No	Default `100`
`rounded`	Rounded corners	`boolean`	No	Default `True`
`match_height`	Match column height	`boolean`	No	Default `False`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `False`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:image-overlay', {
  image: media_image,
  heading_level: 'h3'
}) }}
```

> **Worth knowing**

> The heading inside the panel is a real heading element, set with Heading level. Pick a level that fits the page outline rather than leaving every one at the default.

Video `jarvis:video`

Responsive video, an embed URL (YouTube/Vimeo) or a direct video file.

Use it when: A YouTube or Vimeo embed, or a video file you host.

Paste whatever URL you have. A YouTube watch link, a youtu.be share link, a Vimeo page URL, or an already built embed URL all work, because the component normalises them. A direct link to an mp4 renders a native player instead.

Aspect keeps the frame the right shape while staying responsive.

> **A demo video**
>
> 1. Add Video.
> 2. Paste the YouTube URL straight from the address bar.
> 3. Set Accessible title, which becomes the iframe title for screen reader users.
> 4. Leave Aspect at 16x9 unless your source is a different shape.

Props

Prop	Label	Type	Required	Choices and default
`video_url`	Video URL	`string`	Yes	Free text
`title`	Accessible title	`string`	No	Free text
`captions_url`	Captions file URL (files only)	`string`	No	Free text
`captions_lang`	Captions language code	`string`	No	Default `en`
`aspect`	Aspect ratio	`string`	No	`16x9`, `4x3`, `21x9` Default `16x9`

Prop	Label	Type	Required	Choices and default
`full_width`	Full width	`boolean`	No	Default `False`
`autoplay`	Autoplay (files only)	`boolean`	No	Default `False`
`muted`	Muted (files only)	`boolean`	No	Default `False`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:video', {
  video_url: 'Video URL'
}) }}
```

> **Worth knowing**
>
> Captions matter. For a self hosted file, supply a WebVTT captions URL. Hosted embeds carry their own captions, so that field applies only to direct files.

Video with Sidebar `jarvis:video-with-sidebar`

A video on one side (media-library or a plain URL) with an eyebrow, title, description and call-to-action buttons alongside. Placeable in Canvas or driven by a View.

Use it when: A video beside a column of supporting text and buttons.

Video With Sidebar pairs a player with a text column: an eyebrow line, a heading, description, and call to action buttons. Video on right flips which side the player sits on.

The URL handling is identical to the Video component, because both use the same shared helper.

> **A product explainer**
>
> 1. Add Video With Sidebar.
> 2. Paste the video URL, or pick a media item.
> 3. Set the Heading and description.
> 4. Add the button text and link.
> 5. Tick Video on right if it suits the page rhythm better.

Props

Prop	Label	Type	Required	Choices and default
`video`	Video		No	Free text
`video_url`	Video URL (fallback)	`string`	No	Free text
`captions_url`	Captions file URL (files only)	`string`	No	Free text
`captions_lang`	Captions language code	`string`	No	Default `en`
`aspect`	Video aspect ratio	`string`	No	`16x9`, `4x3`, `21x9` Default `16x9`

Prop	Label	Type	Required	Choices and default
`video_right`	Video on right	`boolean`	No	Default `False`
`eyebrow`	Eyebrow	`string`	No	Free text
`title`	Title	`string`	No	Free text
`heading_level`	Title level	`string`	No	`h2`, `h3`, `h4` Default `h2`
`description`	Description	`string`	No	Free text
`button1_text`	Primary button text	`string`	No	Free text
`button1_href`	Primary button link	`string`	No	Free text
`button2_text`	Secondary button text	`string`	No	Free text
`button2_href`	Secondary button link	`string`	No	Free text
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `False`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:video-with-sidebar', {
  video: 'Video',
  video_url: 'Video URL (fallback)'
}) }}
```

> **Worth knowing**
>
> This one carries a lot of props. Set the video and heading first, and preview before filling in the rest.

Map `jarvis:map`

Google Map from a one-line address (simple_gmap-style keyless embed) with zoom and a link out.

Use it when: An embedded map for a physical address.

Map takes an address and renders an embedded map at the aspect ratio and zoom you choose. Address is the only required prop.

> **A contact page map**
>
> 1. Add Map.
> 2. Type the full street address.
> 3. Adjust Zoom if the default is too tight or too wide.
> 4. Set Aspect to suit the space you have.

Props

Prop	Label	Type	Required	Choices and default
`address`	Address	`string`	Yes	Free text
`zoom`	Zoom	`integer`	No	Default `14`

Prop	Label	Type	Required	Choices and default
`height`	Map height (px)	`integer`	No	Default `400`
`show_link`	Show Google Maps link	`boolean`	No	Default `True`
`link_text`	Link text	`string`	No	Default `View on Google Maps`
`padding`	Padding	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`padding_size`	Padding size	`string`	No	`small`, `medium`, `large` Default `medium`
`margin`	Margin	`string`	No	`none`, `all`, `top`, `bottom`, `left`, `right` Default `none`
`margin_size`	Margin size	`string`	No	`small`, `medium`, `large` Default `medium`
`vertical_padding`	Vertical padding (30px)	`boolean`	No	Default `True`

No slots. Everything this component shows comes from its props.

CALLING IT FROM A TWIG TEMPLATE

```
{{ include('jarvis:map', {
  address: 'Address'
}) }}
```

> **Worth knowing**
>
> Type the address the way you would say it. The component builds the embed URL, so there is no need to hunt for coordinates.

Accessibility for content editors

The theme handles colour contrast automatically, overlays darken until text passes WCAG, unsafe colour picks fall back to a safe colour, and the settings form warns on failing combinations. What it *can't* do is judge your content. The checklist below covers the failures only an editor can prevent:

- **One H1 per page.** The page title (or the hero's heading at level H1) is the only H1. Additional heroes and section headings use H2.
- **Don't skip heading levels.** H2 → H3 → H4, in order. The Hero, CTA and Card components all have a "Heading level" option, set it to one level below the heading that precedes them. Screen-reader users navigate by heading outline; a skipped level is a broken table of contents.
- **Write real alternative text.** Describe what the image *communicates* ("Team collaborating around a whiteboard"), not what it is ("image", "photo of..." , the screen reader already announces it's an image). Purely decorative images get *empty* alt text, not a filename.
- **Link text says where it goes.** "Read the 2026 annual report", never "click here" or a bare "read more". Screen-reader users pull up links as a list, out of context.
- **Caption your videos.** YouTube/Vimeo: enable captions on the platform. Directly-uploaded video files: supply a WebVTT (.vtt) file in the Video component's "Captions file URL" option.
- **Run the checker before publishing.** The Editoria11y toggle (bottom-right on every page when you're logged in) flags all of the above in place. Publish with zero flags.
- **Trust the fallbacks.** If a colour you picked "doesn't take" on a coloured section, that's the ADA guard keeping the text readable, pick a colour the section supports instead of fighting it.

Jarvis targets Drupal 11 and 12 on PHP 8.3 or newer. For install and configuration reference, see the rest of `docs/`.