

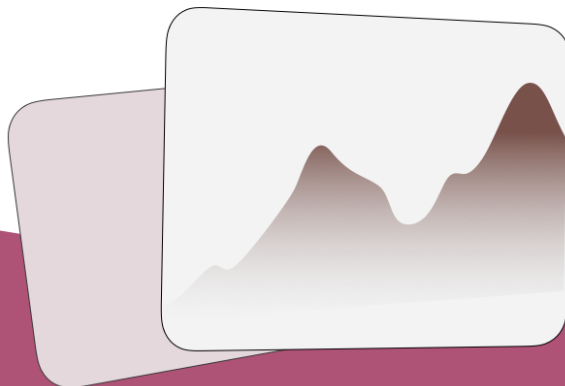
FREE SAMPLE

Swift Charts Beyond the Basics

A practical reference for building advanced data visualizations with the Swift Charts framework

Natalia Panferova & Matthaus Woolard

Released - 2026



Introduction

Thank you so much for purchasing a copy of “Swift Charts Beyond the Basics”. This book is written for developers who already know how to build a basic chart and want to gain a deeper understanding of working with data and the Swift Charts framework.

Swift Charts provides a declarative way to create data visualizations for apps across Apple platforms. When a chart needs to represent more involved data, provide a carefully controlled presentation, or support interaction and accessibility, it becomes important to understand how the framework interprets the content we give it. The structure of the data, the way values are represented, and the mappings between those values and visual properties all influence what the finished chart communicates.

We wrote this book to help you make informed decisions as those requirements become more complex. Instead of focusing only on how to produce a particular result, we explain the framework behavior behind the APIs and how different choices affect the meaning, appearance, and behavior of a chart.

We bring complementary experience to this subject. Natalia has spent many years building mobile apps and worked on UI frameworks at Apple as a member of the core SwiftUI team. Matthaus began his engineering career in data science and has extensive experience building with Swift and Swift Charts, including consulting on applications that work with large and complex datasets. By combining these perspectives, we can examine charts as interactive components in an app and as representations of the data behind them.

The book begins with the data itself. We consider time-series, categorical, proportional, distributional, spatial, and multivariable data, along with the preparation that different relationships and comparisons may require. We then examine how Swift Charts turns prepared values into chart content through marks, plots, grouping, and composition.

From there, we explore how values participate in scales, how chart content is arranged, and how data values become positions, colors, sizes, symbols, and other visual properties. We examine the roles of axes and legends, the relationships among chart layers, and the effects of drawing order, compositing, blend modes, masks, and custom styling.

The later chapters consider charts as interactive parts of an app. We look at how SwiftUI dependencies affect chart updates, how to keep changing state at an appropriate boundary, and how selection, inspection, legends, and scrolling help people explore data. The final chapter brings these ideas together through chart animation, including transitions between plotted values, changing viewports and scale domains, and multiple levels of detail.

Accessibility is part of these decisions throughout the book. We consider semantic descriptions,

differentiation beyond color, contrast, text scaling, reduced transparency, interaction alternatives, and the relationship between a chart's visual and accessible representations where they apply to each subject.

The chapters build on one another, but you can also use the book as a reference and open it at the topic you need. Each section provides the context required to understand its subject while contributing to the broader explanation developed throughout the book.

By the end of the book, you'll have a stronger understanding of how Swift Charts works and how the decisions you make shape the information a chart presents. This will help you approach new datasets and visualization requirements with the knowledge needed to choose and implement an appropriate solution.

The content of this book is copyright Nil Coalescing Limited. It may not be shared or redistributed without prior written permission. If you discover any issues or would like to provide feedback, you can contact support@nilcoalescing.com. Release notes and updates are available at books.nilcoalescing.com/swift-charts-beyond-the-basics.

We hope you enjoy the book!

Arranging chart content

Where chart content appears is determined by more than the horizontal and vertical values supplied to each mark or plot. Swift Charts can divide one position among subcategories, combine several values into a shared stack, or place aggregated observations across a two-dimensional domain. These arrangements determine which observations are compared directly and whether a visible extent represents an individual value, a contribution to a total, or a region of the data.

Some arrangements follow from grouping and stacking values that Swift Charts resolves across the complete composition. Others require us to derive positions from the data, such as mapping recurring times onto a shared reference or converting grid coordinates into cell centers. In both cases, the arrangement needs to retain the original categorical, temporal, or spatial relationships while making the intended comparison possible.

Categorical grouping and stacking establish how several observations can share a primary position or contribute to a common extent. Calendar layouts and heat-map grids extend this positional reasoning to recurring periods and two-dimensional regions. Understanding how each arrangement relates source values to a shared coordinate space lets us control the chart's positions and extents without losing the relationships the data describes.

Categorical grouping

Categorical datasets often describe more than one level of grouping. Several observations may belong to the same primary category while representing different activities, conditions, or series. Placing these subcategories beside one another keeps the larger groups intact and makes their values directly comparable.

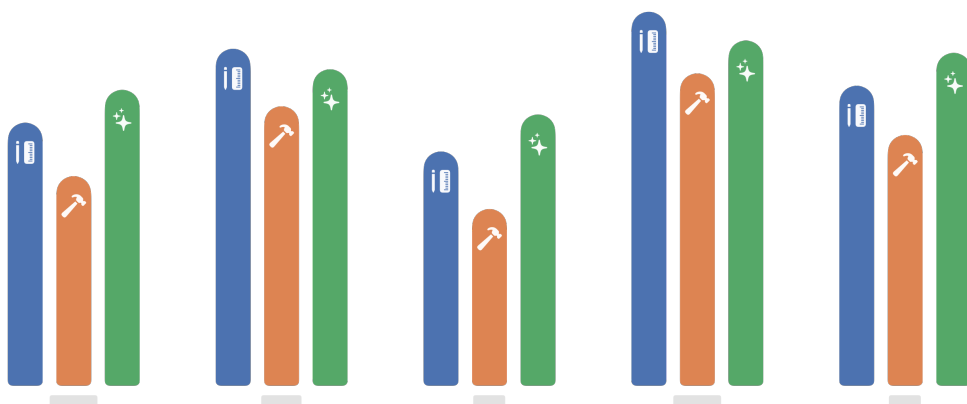
Swift Charts adds this positional encoding with `position(by:axis:span:)`. The modifier uses the subcategory to divide space along the selected axis. When needed, `span` controls the total space available to the positioned content.

Subcategories within primary bands

A categorical scale gives each primary category a band that can be divided among its subcategories. Consider a dataset that records the hours each team member spends on research, design, and development. The following chart groups the activities within each team member's horizontal position:

```
Chart(activityHours) { item in
    BarMark(
        x: .value("Team member", item.teamMember),
        y: .value("Hours", item.hours)
    )
    .foregroundColor(by: .value("Activity", item.activity))
    .position(
        by: .value("Activity", item.activity),
        axis: .horizontal
    )
}
```

Each team member now has three bars, one for each activity. Their heights show the recorded hours, while color repeats the activity encoding so that research, design, and development remain identifiable across the chart.



Grouped bar chart with five categories and three series

The `teamMember` value establishes each horizontal band, and `position(by:axis:)` divides it using `activity`. Swift Charts derives the shared activity order from the first occurrence of each value and reuses it for every team member. Individual groups cannot arrange the same activities in different orders.

Here the grouping runs horizontally because the team members occupy positions along the horizontal axis, so `axis` uses the `horizontal` case. A horizontal bar chart would use the `vertical` case because its primary categories appear along the vertical axis.

Grouping within calendar periods

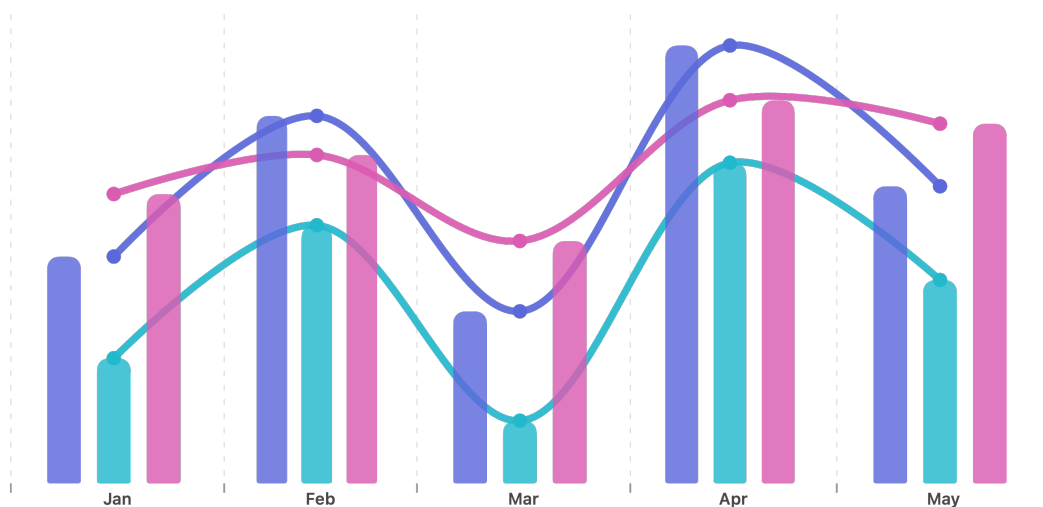
Calendar periods behave differently because dates remain on a continuous scale. Supplying a calendar unit identifies the period represented by each observation, but it does not give every mark a width within that period.

We can see the difference by applying the same monthly values and activity grouping to bars and lines:

```
Chart(monthlyActivityHours) { reading in
    BarMark(
        x: .value("Month", reading.month, unit: .month),
        y: .value("Hours", reading.hours)
    )
    .foregroundStyle(by: .value("Activity", reading.activity))
    .position(
        by: .value("Activity", reading.activity),
        axis: .horizontal
    )

    LineMark(
        x: .value("Month", reading.month, unit: .month),
        y: .value("Hours", reading.hours),
        series: .value("Activity", reading.activity)
    )
    .foregroundStyle(by: .value("Activity", reading.activity))
    .position(
        by: .value("Activity", reading.activity),
        axis: .horizontal
    )
}
```

The activity bars appear beside one another within every month. The three lines continue to pass through the center of each monthly period instead of shifting horizontally to meet their corresponding bars.



Monthly bars divided into subgroup positions while connected series remain centered in each month

BarMark derives an automatic width from the month, giving `position(by:axis:)` a region to divide. Each LineMark contributes only a position at the center of that period, so the activity value does not move the line observation away from its temporal position.

When a temporal comparison needs its subcategories placed side by side, bars and other marks with extent along the temporal axis can use the calendar period directly. Points and connected series remain anchored to their temporal positions even when their dates include a calendar unit.

Positioning compound content

One observation may require several marks that need to move together. A Plot collects those components so that we can apply one positional modifier to the complete representation.

Consider annual water-intake summaries that compare daytime and nighttime measurements. Each summary describes the complete measured range, the range containing the middle half of the values, and the median. We can represent these properties with a rule for the complete range, a rectangle for the middle half, and a narrow rectangle for the median:

```

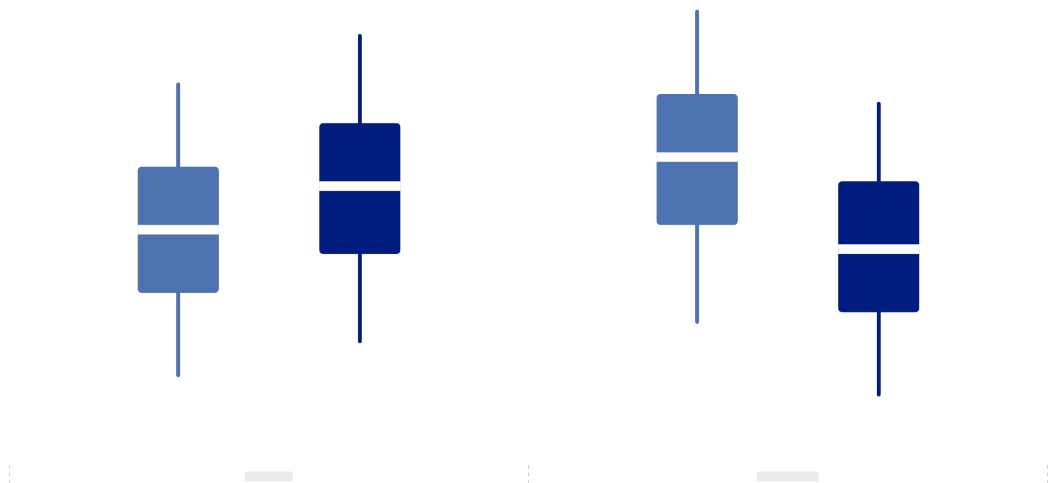
Chart(summaries) { summary in
  Plot {
    RuleMark(
      x: .value("Year", summary.year, unit: .year),
      yStart: .value("Minimum water intake", summary.minimum),
      yEnd: .value("Maximum water intake", summary.maximum)
    )

    RectangleMark(
      x: .value("Year", summary.year, unit: .year),
      yStart: .value("Lower quartile", summary.lowerQuartile),
      yEnd: .value("Upper quartile", summary.upperQuartile),
      width: .ratio(0.7)
    )

    RectangleMark(
      x: .value("Year", summary.year, unit: .year),
      yStart: .value(
        "Median lower edge",
        summary.medianLowerEdge
      ),
      yEnd: .value(
        "Median upper edge",
        summary.medianUpperEdge
      ),
      width: .ratio(0.7)
    )
  }
  .foregroundStyle(by: .value("Day or night", summary.dayOrNight))
  .position(
    by: .value("Day or night", summary.dayOrNight),
    axis: .horizontal
  )
}

```

The daytime and nighttime box plots appear beside one another within each year. Every whisker, quartile range, and median stays aligned with the other parts of its box plot.



Paired vertical box plots

The `year` value associates each summary with an annual period, and `dayOrNight` selects its position within that period. Because `foregroundStyle(by:)` and `position(by:axis:)` are attached to the surrounding `Plot`, Swift Charts styles and moves all three marks together. We can also apply an accessibility label and value to the same `Plot` so that accessibility technologies present the summary as one observation.

Positional grouping works at the level where we apply the modifier. On an individual bar, it selects a position within a categorical band or calendar period. On a `Plot`, the same encoding positions the complete compound observation while preserving the relationships among its components.

Stacking

Stacking combines several bar or area values along a measured axis so that every value contributes to a shared extent. Unlike positional grouping, which assigns subcategories separate positions within a band, stacking keeps related values at the same position and accumulates them along the other axis.

Bars and areas express this cumulative relationship in different ways. Bars accumulate discrete values that occupy the same position, while area stacks accumulate connected series across a sequence of shared positions.

Stacking methods

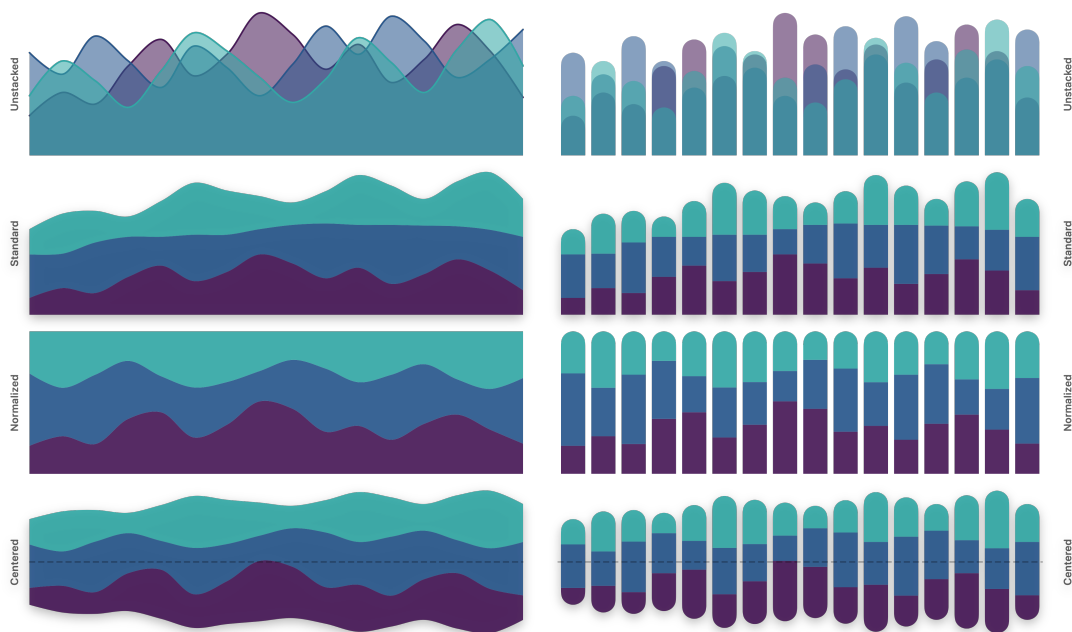
`MarkStackingMethod` provides four ways to relate these values: `unstacked`, `standard`, `normalized`, and `center`. Swift Charts uses `standard` by default when a bar or area supplies one value along the stacking axis. We can select another method through the `stacking` parameter in both mark and vectorized plot declarations. In a vectorized plot, the selected method applies to the complete collection and cannot vary between its elements.

This area chart selects `center` stacking for power-draw readings from several devices:

```
ForEach(readings) { reading in
    AreaMark(
        x: .value("Time", reading.timestamp),
        y: .value("Power draw", reading.powerDraw),
        series: .value("Device", reading.device),
        stacking: .center
    )
    .foregroundColor(
        by: .value("Device", reading.device)
    )
}
```

The `series` value determines which readings form each connected area. The stacking method then controls how those areas share the vertical extent at every timestamp, while foreground style keeps the device series visually distinguishable.

Applying the same source values with each method shows how stacking changes both area and bar content:



Four rows comparing unstacked, standard, normalized, and centered stacking for area charts and bar charts

With `unstacked`, each area or bar retains its own zero baseline, so values that share a position overlap. `standard` instead builds a cumulative extent by placing each segment after the preceding one. This preserves the original magnitudes while making the outer boundary represent their total.

When relative contribution matters more than total magnitude, `normalized` converts the values at each position into shares of their total. Every complete stack then has a consistent extent, allowing changes in proportion to remain distinct from changes in the overall amount. The `center` method also preserves the original magnitudes, but offsets the cumulative extent around a centered baseline, creating a streamgraph when used with areas.

An area stack requires every series to provide a value at each shared position. Without corresponding values, Swift Charts cannot construct the complete cumulative extent at that position.

Chart-wide area stacks

A `Plot` groups chart content structurally, but it does not establish the boundaries of an area stack. Swift Charts resolves area stacking across the surrounding `Chart`, so area series in separate `Plot` containers can still contribute to the same cumulative extent.

This distinction matters when separate `Plot` groups use the same stacking method. Consider one collection of power readings grouped by component and another grouped by sensor:

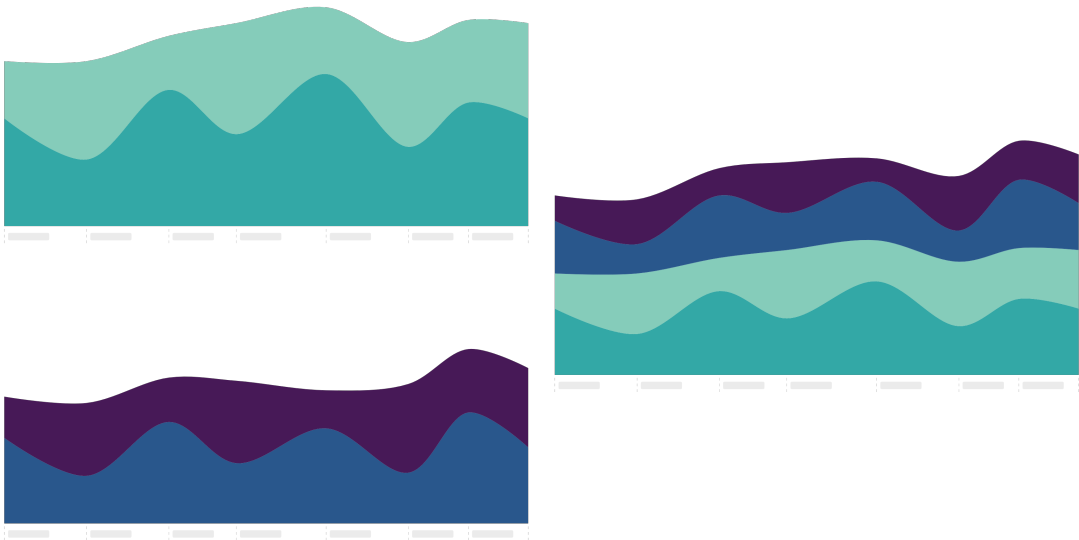
```

Plot {
  AreaPlot(
    componentPowerReadings,
    x: .value("Time", \.time),
    y: .value("Power draw", \.powerDraw),
    series: .value("Component", \.component),
    stacking: .standard
  )
}

Plot {
  AreaPlot(
    fanPowerReadings,
    x: .value("Time", \.time),
    y: .value("Power draw", \.powerDraw),
    series: .value("Sensor", \.sensor),
    stacking: .standard
  )
}

```

Placing each Plot in its own Chart creates two independent two-series stacks. Placing both in one Chart combines all four series into a single stack:



Two separate standard-stacked area charts beside the same area plots combined into one chart

Loops, nested chart content, and custom ChartContent types behave the same way: they can organize a composition, but they do not establish stacking boundaries. The order within the combined stack follows the order in which each series first appears in the composition.

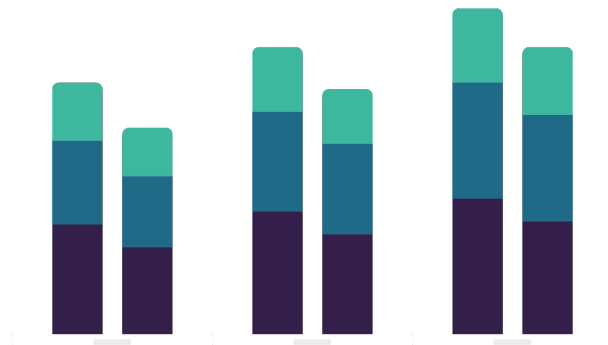
Grouped bar stacks

Positional grouping and stacking can describe two levels of categorical structure within the same bar chart. `position(by:axis:)` divides a primary band into subgroup positions, and bars at each resulting position form their own stack.

Suppose a power-draw dataset records the contribution of several usage types during the day and night for each residence:

```
BarMark(
  x: .value("Residence", reading.residence),
  y: .value("Power draw", reading.powerDraw),
  stacking: .standard
)
.foregroundColor(
  by: .value("Usage type", reading.usageType)
)
.position(
  by: .value("Day or night", reading.dayOrNight),
  axis: .horizontal
)
```

The chart gives every residence separate daytime and nighttime bars. The height of each bar represents its total power draw, while the colored segments show how much each usage type contributes to that total.



Power draw for residences grouped by day and night, with each bar stacked by usage type

The residence value establishes the primary horizontal bands, and `dayOrNight` selects one of two positions within each band. At every resulting position, `standard` stacking accumulates the `powerDraw` values, while foreground style identifies the usage type represented by each segment.

Applying positional grouping to area content does not create the same independent stacking boundaries. Area series continue to participate in the chart-wide stack determined by their stacking method.

Independent area stacks

When several area groups need separate baselines within the same coordinate space, their chart-wide stacking behavior means that automatic stacking cannot produce the required arrangement. We instead need to calculate the cumulative lower and upper bounds of every area ourselves.

Assuming group is nonempty and each series stores one reading for every shared position in the same order, the following calculation processes the group in the intended series order:

```
var stackedReadings: [StackedReading] = []

for position in group[0].readings.indices {
    var cumulativeValue = 0.0

    for series in group {
        let reading = series.readings[position]
        let stackStart = cumulativeValue
        cumulativeValue += reading.value

        stackedReadings.append(
            StackedReading(
                time: reading.time,
                name: series.name,
                value: reading.value,
                stackStart: stackStart,
                stackEnd: cumulativeValue
            )
        )
    }
}
```

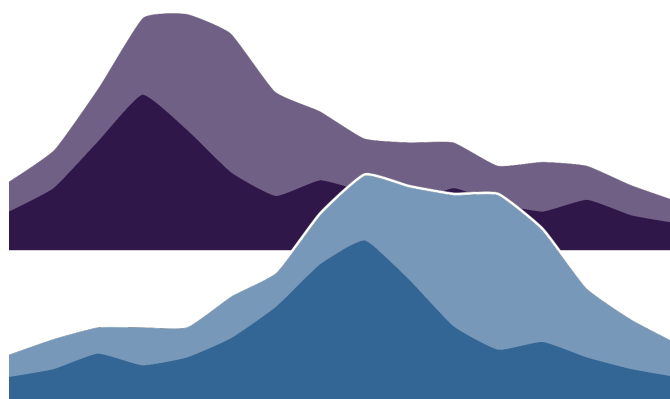
The outer loop uses the readings of the first series to visit every shared position. At each position, the inner loop follows the chosen series order and advances `cumulativeValue`. Every resulting `StackedReading` retains the original value as well as the cumulative `stackStart` and `stackEnd` used to position its area.

After applying the same calculation to both groups, we pass their explicit bounds to separate `AreaPlot` declarations:

```
AreaPlot(
    stackedReadings,
    x: .value("Time", \.time),
    yStart: .value("Lower power bound", \.stackStart),
    yEnd: .value("Upper power bound", \.stackEnd),
    series: .value("Residence", \.name)
)
.accessibilityLabel(\.accessibilityLabel)
.accessibilityValue(\.accessibilityValue)
.offset(y: -ridgeOffset)

AreaPlot(
    otherStackedReadings,
    x: .value("Time", \.time),
    yStart: .value("Lower power bound", \.stackStart),
    yEnd: .value("Upper power bound", \.stackEnd),
    series: .value("Residence", \.name)
)
.accessibilityLabel(\.accessibilityLabel)
.accessibilityValue(\.accessibilityValue)
```

Because `yStart` and `yEnd` describe the area bounds directly, the two plots no longer participate in automatic stacking. Applying `offset(y: )` to the first group moves it upward, allowing the independent stacks to overlap as separate ridges.



Paired area ridges positioned with explicit lower and upper bounds

The custom accessibility values should describe the original power-draw measurements rather than the cumulative bounds used to arrange them. Those bounds are derived layout values and do not replace the meaning of the observations.

Automatic stacking is appropriate when bars or area series should participate in one shared cumulative relationship. Explicit bounds give us control when separate area groups need independent cumulative extents within the same chart, but they also make the application responsible for their alignment, order, and accessible descriptions.

Calendar layouts

Calendar layouts arrange observations by deriving two positions from their temporal values. One axis can retain an observation's place in a larger calendar sequence, while the other maps a recurring component, such as its time of day or weekday, onto a shared reference period.

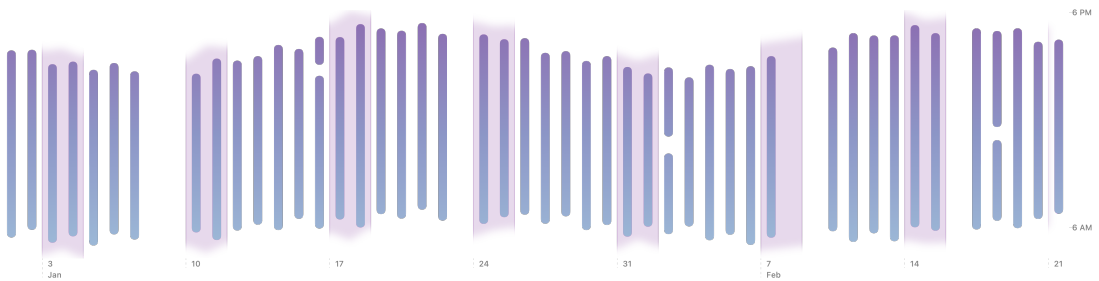
These derived dates make recurring intervals and calendar components directly comparable, but they don't replace the source dates that give the observations their meaning. We therefore need to preserve the original dates, durations, and calendar relationships when describing the chart's values.

Recurring time ranges

A temporal interval can be located both by the calendar period it belongs to and by the part of a recurring cycle that it occupies. We can encode the period on one axis and the interval on the other. The second axis uses a shared reference date so that equivalent times occupy the same positions across observations.

After reconstructing each start time on the reference date, we derive its end by adding the interval's original duration. This keeps an interval that crosses the cycle boundary continuous instead of placing its end before its start.

Consider a collection of sleep sessions. Each session belongs to a particular night and spans an interval within the daily cycle. Plotting the night horizontally and the normalized time range vertically lets us compare changes in both the beginning and end of each session without reducing the interval to a total duration.



Range bars comparing the start and end times of nightly sleep sessions

Each vertical bar extends from the time when a session began to the time when it ended. Because every session uses the same vertical reference period, we can compare changes in its start and end times, while the length of the bar continues to represent its duration.

We first need to decide which night each session belongs to. Using midday as the boundary

between consecutive nights, we can subtract twelve hours from the start time:

```
let sleepDay = calendar.date(
    byAdding: .hour, value: -12, to: sleep.startDate
)!
```

The chart groups `sleepDay` by its calendar day. A session beginning before noon moves into the preceding day, while a session beginning after noon remains in its current day. This keeps periods of sleep separated by a brief awakening after midnight within the same night.

The vertical positions need a different form of normalization. We reconstruct the session's start time on a shared reference day, then add its original duration to obtain the end:

```
let startComponents = calendar.dateComponents(
    [.hour, .minute, .second],
    from: sleep.startDate
)

let referenceDate = calendar.startOfDay(for: Date())
let normalizedStart = calendar.date(
    bySettingHour: startComponents.hour!,
    minute: startComponents.minute!,
    second: startComponents.second!,
    of: referenceDate
)

let normalizedEnd = normalizedStart.addingTimeInterval(
    sleep.endDate.timeIntervalSince(sleep.startDate)
)
```

Every session now begins within the same reference day. When a session continues across midnight, `normalizedEnd` falls on the following day, preserving the continuous interval instead of wrapping its end back to the beginning of the reference day.

The assigned night and normalized endpoints provide the positions for the range bar:

```
BarMark(
    x: .value("Day", sleepDay, unit: .day, calendar: calendar),
    yStart: .value("Sleep start time", normalizedStart),
    yEnd: .value("Sleep end time", normalizedEnd)
)
```

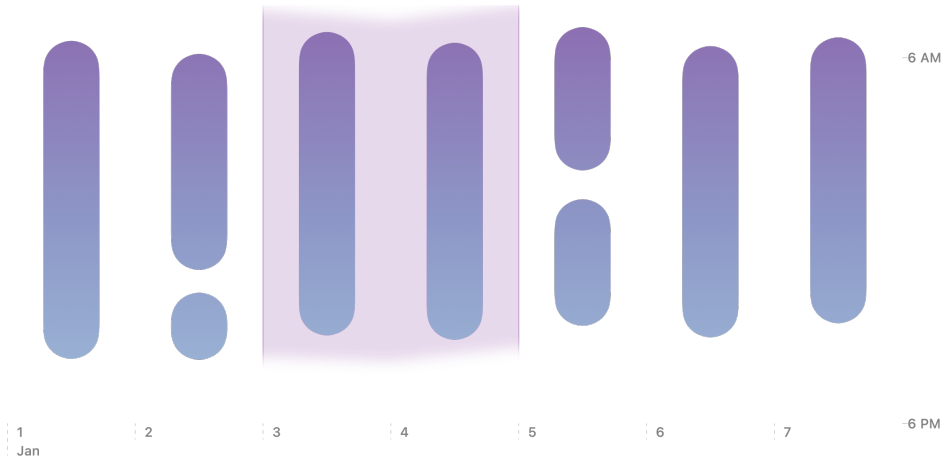
The normalized dates are useful for layout, but their reference day is not part of the underlying observation. We can replace the bar's automatic accessibility description with the original temporal meaning:

```
.accessibilityLabel(
    Text("Sleep on \((sleepDay, format: .dateTime.month().day())")
)
.accessibilityValue(
    Text(
        """
        \((sleep.startDate, format: .dateTime.hour().minute()) to \
        \((sleep.endDate, format: .dateTime.hour().minute())
        """
    )
)
```

The label identifies the night represented by the bar, while the value describes its original start

and end times. The chart can therefore use normalized dates for comparison without presenting their shared reference day as part of the observation.

With several nights in the chart, the default vertical scale places earlier times toward the bottom and later times toward the top.

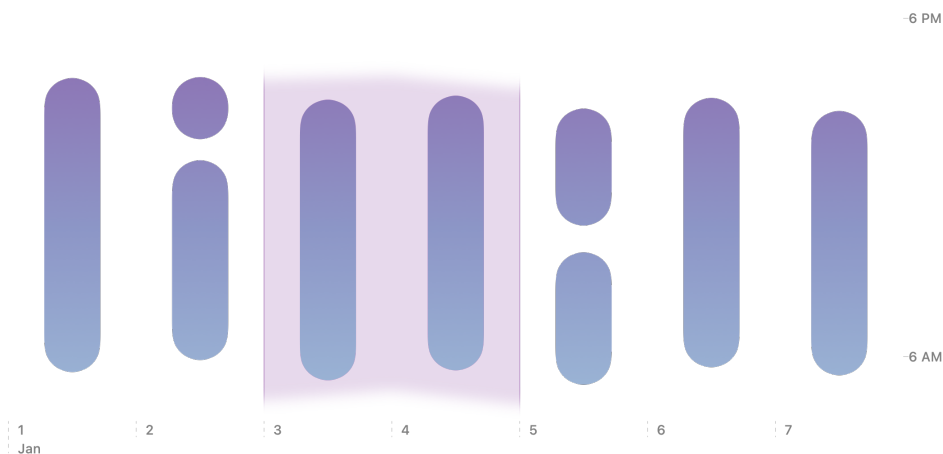


Seven nightly sleep intervals with two awake gaps and time increasing upward from evening to morning

For this schedule-like arrangement, placing earlier times at the top and later times at the bottom follows the progression of each night more naturally. We can reverse the vertical scale while retaining its automatic domain:

```
Chart {  
  // ... sleep interval marks ...  
}  
.chartYScale(  
  domain: .automatic(reversed: true),  
  range: .plotDimension(startPadding: 12, endPadding: 12)  
)
```

The `reversed` argument changes the direction of the inferred y-axis domain without requiring us to calculate its bounds. The `startPadding` and `endPadding` values keep the earliest and latest endpoints away from the edges of the plot area.



Seven nightly sleep intervals with two awake gaps and time descending from evening to morning

Calendar grids

A calendar grid applies the same reference-period technique to individual dates. It arranges each observation by two components of the same date: its week within the calendar sequence and its weekday within that week. Aligning matching weekdays in columns while preserving the sequence of weeks in rows makes daily observations comparable across both dimensions.

We can create this arrangement by plotting the original date vertically by week and a normalized weekday horizontally:

```
let normalizedWeekday = dataPoint.date.normalizedToReferenceWeek(in: calendar)

RectangleMark(
  x: .value(
    "Day of week", normalizedWeekday,
    unit: .weekday, calendar: calendar
  ),
  y: .value(
    "Week of year", dataPoint.date,
    unit: .weekOfYear, calendar: calendar
  )
)
.foregroundColor(by: .value("Value", dataPoint.value))
```

The original date supplies the vertical week position. Its normalized counterpart supplies the horizontal weekday position, while foreground style maps the observation's value to color.

The normalized date retains the weekday and time components while replacing the original week with a shared reference week:

```

extension Date {
    func normalizedToReferenceWeek(in calendar: Calendar) → Date {
        let referenceDate = Date(timeIntervalSinceReferenceDate: 0)

        var components = calendar.dateComponents(
            [.yearForWeekOfYear, .weekOfYear],
            from: referenceDate
        )

        let recurringComponents = calendar.dateComponents(
            [.weekday, .hour, .minute, .second],
            from: self
        )

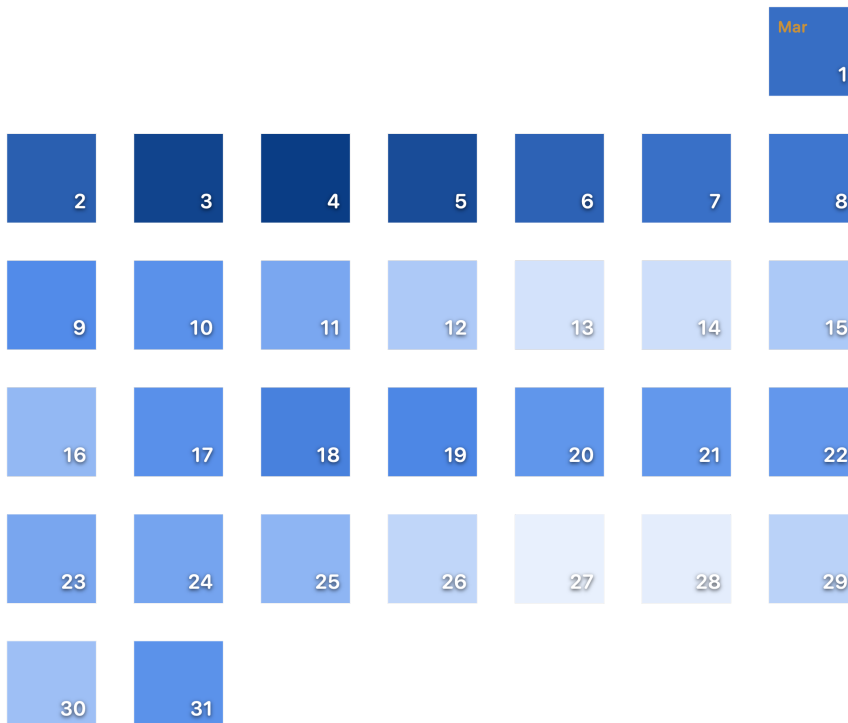
        components.weekday = recurringComponents.weekday
        components.hour = recurringComponents.hour
        components.minute = recurringComponents.minute
        components.second = recurringComponents.second

        return calendar.date(from: components)!
    }
}

```

Every occurrence of the same weekday now belongs to the same horizontal calendar period. Passing the same `Calendar` to the normalization and plotting operations keeps the weekday and week-of-year relationships consistent with the calendar used by the chart.

Rendering one month of daily observations produces the basic grid.



Month boundaries

A `RectangleMark` centered on unit-based temporal values uses automatic dimensions that leave space between neighboring marks. We can instead derive its width and height from the complete weekday and week periods, then reduce those dimensions by fixed screen-space insets.

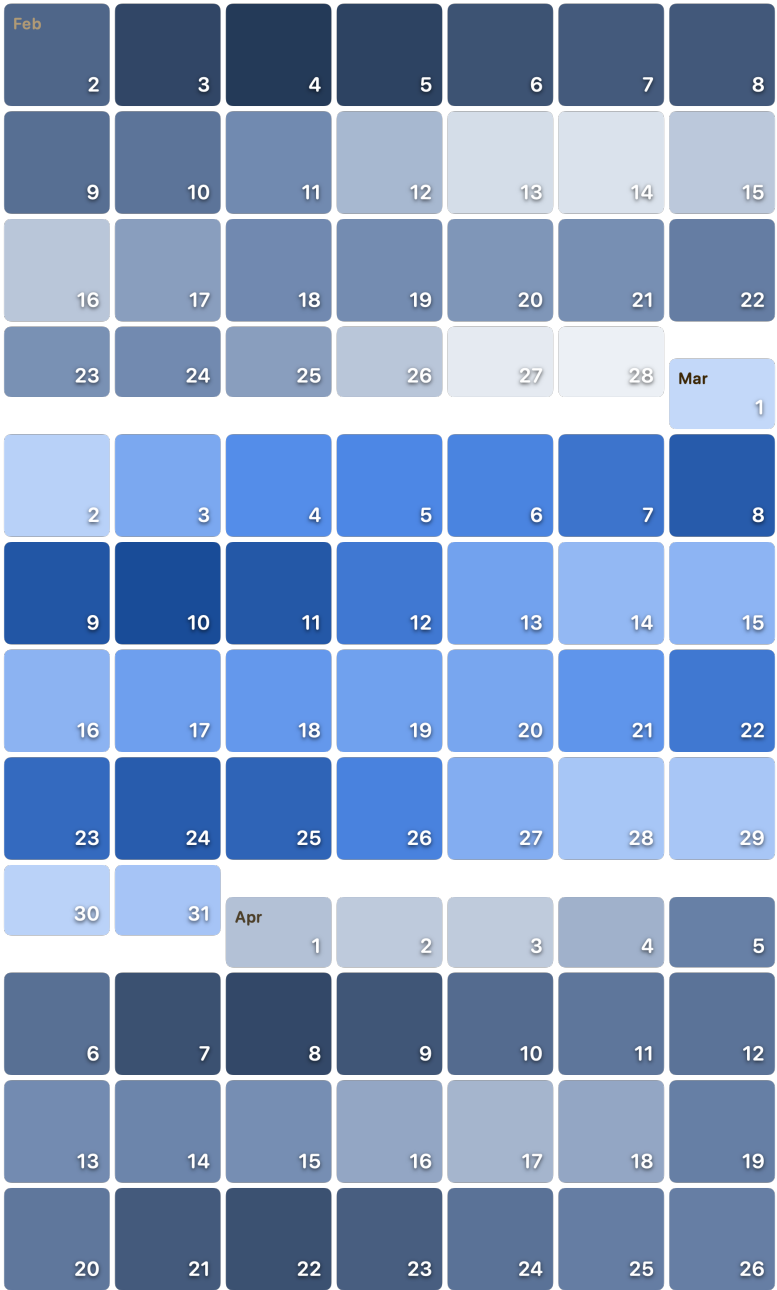
A week that contains the end of one month and the beginning of another requires additional separation. Assuming each data point records whether its date belongs to the first or last week of its month, a larger vertical inset and opposite offsets separate the two parts of the shared week:

```
let normalizedWeekday = dataPoint.date.normalizedToReferenceWeek(in: calendar)
let isMonthBoundary = dataPoint.isInFirstWeekOfMonth
    || dataPoint.isInLastWeekOfMonth

let heightInset: CGFloat = isMonthBoundary ? 14 : 2
let verticalOffset: CGFloat = dataPoint.isInFirstWeekOfMonth
    ? 12
    : dataPoint.isInLastWeekOfMonth ? -12 : 0

RectangleMark(
    x: .value(
        "Day of week", normalizedWeekday,
        unit: .weekday, calendar: calendar
    ),
    y: .value(
        "Week of year", dataPoint.date,
        unit: .weekOfYear, calendar: calendar
    ),
    width: .inset(2),
    height: .inset(heightInset)
)
.foregroundColor(by: .value("Value", dataPoint.value))
.offset(y: verticalOffset)
```

Most cells use a two-point inset along both dimensions. Cells in a month-boundary week use a larger height inset, while the offset moves dates from the ending month upward and dates from the beginning month downward. Together, these adjustments create a visible division between the months that share the same week row.



Calendar heat map with days arranged by week and weekday

The insets and offsets alter the marks in screen space without replacing their plotted dates. Accessibility navigation can therefore continue to follow the chart’s weekday and week-of-year axes.

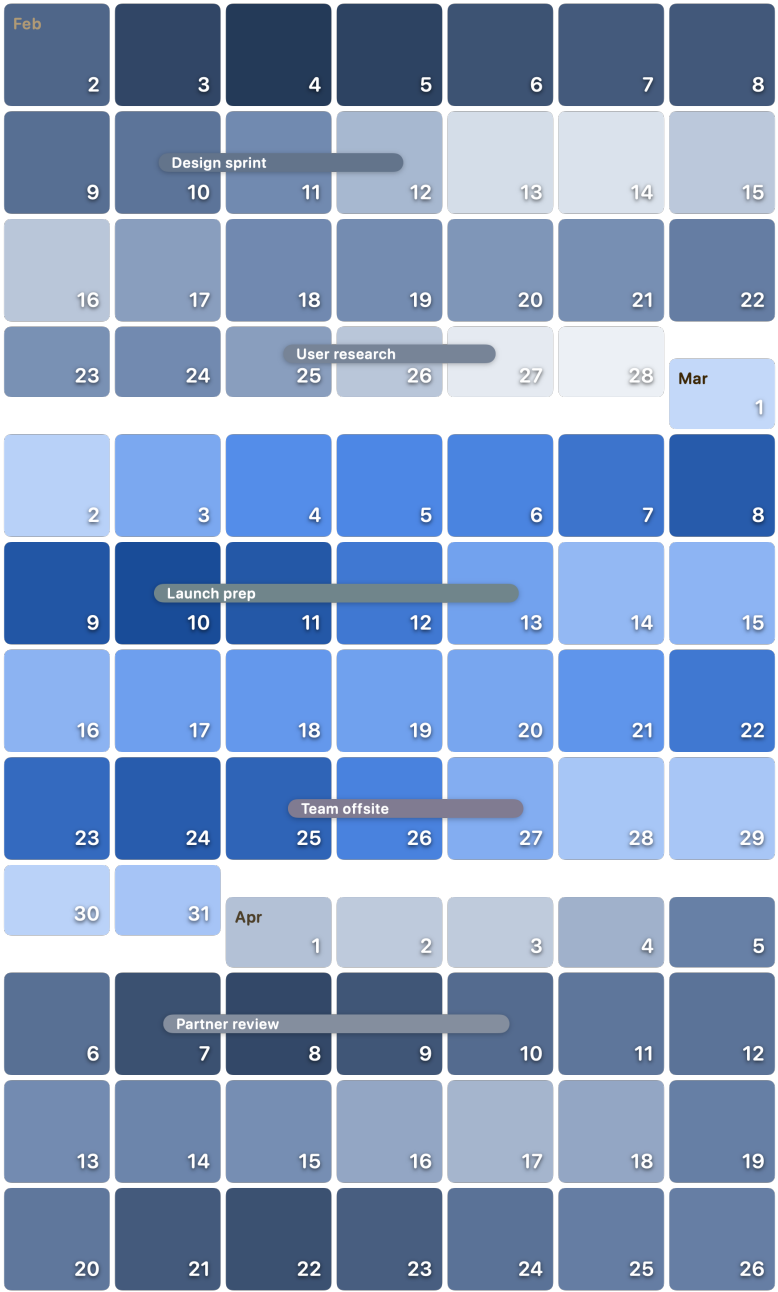
Multi-day events

The normalized weekday scale can also position intervals within a week. For an event whose start and end belong to the same calendar week, we provide its normalized endpoints to a horizontal BarMark and use the original start date to select the vertical week:

```
let normalizedStart = event.startTime.normalizedToReferenceWeek(in: calendar)
let normalizedEnd = event.endTime.normalizedToReferenceWeek(in: calendar)

BarMark(
    xStart: .value("From", normalizedStart),
    xEnd: .value("Until", normalizedEnd),
    y: .value(
        "Week of year", event.startTime,
        unit: .weekOfYear, calendar: calendar
    ),
    height: .fixed(14)
)
.accessibilityLabel(
    Text("\(event.title), starting \(event.startTime, format: startDayFormat)")
)
.accessibilityValue(
    Text("Duration: \(event.eventDuration, format: durationFormat)")
)
```

The bar begins and ends at the appropriate weekday and time within the reference week. Its vertical value places it in the week in which the event occurs. Because the event bars and calendar cells use the same normalized weekday scale, their positions align directly.



Calendar heat map with multi-day event bars arranged by weekday and week

The custom accessibility label and value retain the event’s original start date and duration instead of exposing the reference week used for layout. An event that crosses a calendar-week boundary cannot remain within one week row, so it needs to be divided into separate intervals before being placed in the grid.

Heat-map grids

Some datasets associate an aggregate with each region of a two-dimensional domain. A heat map preserves the spatial arrangement of those regions and uses a visual encoding, commonly color, to represent the value calculated for each one. The geometry needed to position a region depends on the shape of its grid cell.

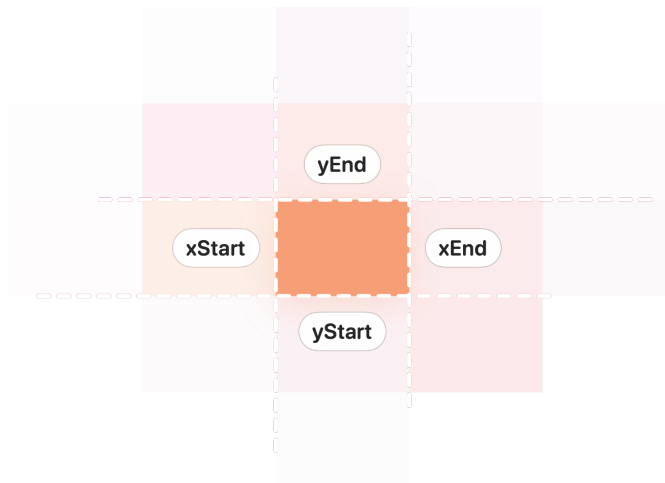
Rectangular heat-map cells

A rectangular heat-map cell represents the intersection of one range on the horizontal axis and another on the vertical axis. When observations have been grouped with `NumberBins` along both dimensions, each aggregate contains horizontal and vertical bin ranges together with a value calculated from the observations within their intersection.

Consider product observations grouped into price ranges and sales-volume ranges, with total sales calculated for every combination:

```
// Each cell contains product-price and sales-volume ranges from
// NumberBins, plus the total sales aggregated within those ranges.
RectangleMark(
  xStart: .value("Product price from", cell.priceRange.lowerBound),
  xEnd: .value("Product price to", cell.priceRange.upperBound),
  yStart: .value("Sales volume from", cell.salesVolumeRange.lowerBound),
  yEnd: .value("Sales volume to", cell.salesVolumeRange.upperBound)
)
.foregroundColor(by: .value("Total sales", cell.totalSales))
```

The `xStart` and `xEnd` values place the left and right edges of the cell, while `yStart` and `yEnd` place its lower and upper edges. These four values determine the rectangle's position and extent. The foreground style independently maps `totalSales` to color. The following illustration isolates one cell and the four boundaries that define it:



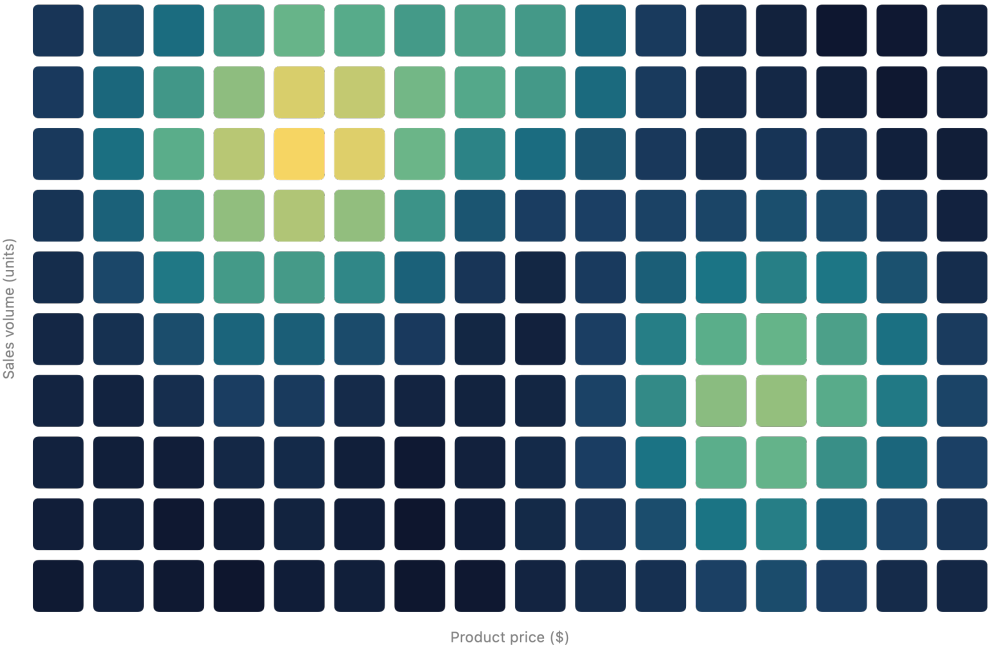
Heat map with a highlighted rectangular cell labeled `xStart`, `xEnd`, `yStart`, and `yEnd`

Because neighboring bins share their boundary values, their rectangles meet without leaving gaps. We can create visible separation between the cells in screen space without changing the data ranges they represent.

For a vectorized `RectanglePlot`, the `PlottableProjection.value(_:_:_)` overload supplies the lower and upper bounds as one projection for each axis. This form also lets us provide inset dimensions for the complete horizontal and vertical ranges:

```
RectanglePlot(  
    cells,  
    x: .value(  
        "Product price",  
        \.priceRange.lowerBound, \.priceRange.upperBound  
    ),  
    y: .value(  
        "Sales volume",  
        \.salesVolumeRange.lowerBound,  
        \.salesVolumeRange.upperBound  
    ),  
    width: .inset(1),  
    height: .inset(1)  
)  
.foregroundStyle(  
    by: .value("Total sales", \.totalSales)  
)  
.cornerRadius(4)
```

Each projection retrieves both boundary values for a cell. `RectanglePlot` derives the cell's complete width and height from those values, then `inset(1)` removes one point from each side in screen coordinates. The corner radius rounds the resulting rectangle. These adjustments distinguish neighboring cells visually without changing their encoded ranges or aggregated values.



Heat map of product price and sales-volume ranges, with color representing total sales

The visible gaps make the individual price and sales-volume regions easier to distinguish, while color continues to show how total sales vary across the grid.

Nonrectangular grids

Cells whose shapes cannot be described by horizontal and vertical ranges need a different representation. Instead of supplying the boundaries of each region directly, we can position a point at the cell’s center and use a custom symbol to provide its visible shape.

Consider a spatial summary that groups recent earthquake observations into hexagonal cells. Each cell records the number of earthquakes assigned to it and is identified by axial `q` and `r` coordinates stored in `HexCell.ID`. These coordinates locate the cell within the hexagonal grid. As part of preparing the data, the grid converts them back into the latitude and longitude of the cell’s geographic center.

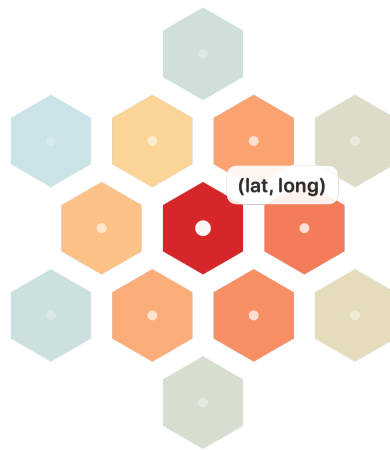
The following chart uses that center to position a `PointMark` for each cell:

```

PointMark(
  x: .value("Longitude", cell.center.longitude),
  y: .value("Latitude", cell.center.latitude)
)
.foregroundColor(
  by: .value("Recent earthquakes", cell.recentEarthquakes)
)
.symbol {
  Hexagon()
}

```

The longitude and latitude values anchor the geographic center of the cell. The application-specific Hexagon shape supplies its visible region around that position, while the foreground style maps the aggregated earthquake count to color. The following illustration separates these responsibilities by showing the center positions within the hexagonal regions:



Hexagonal grid with a selected central cell labeled at its center and surrounding cells marked at their centers

Using the geographic center of every occupied cell preserves the spatial relationships established by the grid. Placing those cells over the corresponding geographic context produces the following map:



Hexagonal heat map of illustrative earthquake density across New Zealand

The chart also needs an accessible description of the region and the aggregate represented by its color. If the prepared data associates each cell with a recognizable `regionName`, we can add the corresponding modifiers to `PointMark`:

```
.accessibilityLabel(
  Text("Recent earthquakes near \$(cell.regionName)")
)
.accessibilityValue(
  Text(cell.recentEarthquakes, format: .number)
)
```

The accessible representation now identifies each cell by a meaningful geographic region and reports the earthquake count encoded by its color.

By matching each mark's positioning strategy to the geometry established during data preparation, we can preserve the grid's spatial relationships while giving every aggregate an appropriate visual and accessible representation.