Package 'bidux'

February 27, 2026

Title Behavioral Insight Design: A Toolkit for Integrating Behavioral Science in UI/UX Design

Version 0.4.0

Description Provides a framework and toolkit to guide R dashboard developers in implementing the Behavioral Insight Design (BID) framework. The package offers functions for documenting each of the five stages (Interpret, Notice, Anticipate, Structure, and Validate), along with a comprehensive concept dictionary. Works with both 'shiny' applications and 'Quarto' dashboards.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1.0)

RoxygenNote 7.3.3

Imports cli, DBI, dplyr, glue, janitor, jsonlite, memoise, readr (>= 2.1.5), rlang, RSQLite, stats, stringdist (>= 0.9.15), tibble (>= 3.2.1), tools, utils

Suggests DiagrammeR, knitr, otel, otelsdk, rmarkdown, spelling, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Language en-US

URL https://jrwinget.github.io/bidux/

BugReports https://github.com/jrwinget/bidux/issues

NeedsCompilation no

Author Jeremy Winget [aut, cre] (ORCID: <https://orcid.org/0000-0002-3783-4354>)

Maintainer Jeremy Winget <contact@jrwinget.com>

Repository CRAN

Date/Publication 2026-02-27 22:02:06 UTC

Contents

`as_tibble.bid_issues` *Convert bid_issues object to tibble*

Description

Extracts the tidy issues tibble from a bid_issues object for analysis and visualization. This provides a structured view of all telemetry issues with metadata for prioritization and reporting.

Usage

```
## S3 method for class 'bid_issues'
as_tibble(x, ...)
```

Arguments

`x`	A bid_issues object from bid_ingest_telemetry()
`...`	Additional arguments (unused)

Value

A tibble with issue metadata including severity, impact, and descriptions

---

`as_tibble.bid_stage` *Convert bid_stage to tibble*

Description

Convert bid_stage to tibble

Usage

```
## S3 method for class 'bid_stage'
as_tibble(x, ...)
```

Arguments

`x`	A bid_stage object
`...`	Additional arguments (unused)

Value

A tibble

bid_address *Create Notice stage from single telemetry issue (sugar)*

Description

Convenience function that combines issue selection and Notice creation in one step. Useful for quick workflows where you want to address a specific issue immediately.

Usage

```
bid_address(issue, previous_stage, ...)
```

Arguments

issue	A single row from bid_telemetry() output
previous_stage	Previous BID stage (typically from bid_interpret)
...	Additional arguments passed to bid_notice_issue()

Value

A bid_stage object in the Notice stage

Examples

```
## Not run:
issues <- bid_telemetry("data.sqlite")
interpret <- bid_interpret("How can we improve user experience?")

# Address the highest impact issue
top_issue <- issues[which.max(issues$impact_rate), ]
notice <- bid_address(top_issue, interpret)

## End(Not run)
```

bid_anticipate *Document User Behavior Anticipation Stage in BID Framework*

Description

This function documents the anticipated user behavior by listing bias mitigation strategies related to anchoring, framing, confirmation bias, etc. It also supports adding interaction hints and visual feedback elements.

Usage

```
bid_anticipate(
  previous_stage,
  bias_mitigations = NULL,
  include_accessibility = TRUE,
  quiet = NULL,
  ...
)
```

Arguments

previous_stage	A tibble or list output from an earlier BID stage function.
bias_mitigations	A named list of bias mitigation strategies. If NULL, the function will suggest bias mitigations based on information from previous stages.
include_accessibility	Logical indicating whether to include accessibility mitigations. Default is TRUE.
quiet	Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).
...	Additional parameters. If 'interaction_principles' is provided, it will be ignored with a warning.

Value

A tibble containing the documented information for the "Anticipate" stage.

Examples

```
interpret_stage <- bid_interpret(
  central_question = "How can we improve selection efficiency?",
  data_story = new_data_story(
    hook = "Too many options",
    context = "Excessive choices",
    tension = "User frustration",
    resolution = "Simplify menu"
  )
)

notice_stage <- bid_notice(
  previous_stage = interpret_stage,
  problem = "Issue with dropdown menus",
  evidence = "User testing indicated delays"
)

structure_info <- bid_structure(previous_stage = notice_stage)

# Let the function suggest bias mitigations based on previous stages
bid_anticipate(previous_stage = structure_info)
```

```
# with accessibility included (default) and custom bias mitigations
anticipate_result <- bid_anticipate(
  previous_stage = structure_info,
  bias_mitigations = list(
    anchoring = "Use context-aware references",
    framing = "Toggle between positive and negative framing"
  ),
  include_accessibility = TRUE
)

summary(anticipate_result)
```

---

bid_concept *Get detailed information about a specific concept*

Description

Returns detailed information about a specific BID framework concept, including implementation recommendations based on the concept's stage.

Usage

```
bid_concept(concept_name, add_recommendations = TRUE)
```

Arguments

concept_name	A character string with the exact or partial concept name
add_recommendations	Logical indicating whether to add stage-specific recommendations

Value

A tibble with detailed concept information

---

bid_concepts *Search BID Framework Concepts*

Description

Search for behavioral science and UX concepts used in the BID framework. Returns concepts matching the search term along with their descriptions, categories, and implementation guidance.

Usage

```
bid_concepts(search = NULL, fuzzy_match = TRUE, max_distance = 2)
```

Arguments

search	A character string to search for. If NULL or empty, returns all concepts.
fuzzy_match	Logical indicating whether to use fuzzy string matching (default: TRUE)
max_distance	Maximum string distance for fuzzy matching (default: 2)

Value

A tibble containing matching concepts with their details

---

bid_flags *Extract telemetry flags from bid_issues object*

Description

Extracts global telemetry flags and metadata from a bid_issues object. These flags provide boolean indicators for different types of issues and can be used for conditional logic in downstream BID stages.

Usage

```
bid_flags(x)

## S3 method for class 'bid_issues'
bid_flags(x)

## Default S3 method:
bid_flags(x)
```

Arguments

x	A bid_issues object from bid_ingest_telemetry() or any object with a flags attribute

Value

A named list of boolean flags and metadata

bid_get_quiet *Get current quiet mode setting*

Description

Check whether bidux is currently in quiet mode.

Usage

```
bid_get_quiet()
```

Value

Logical indicating whether quiet mode is enabled

Examples

```
# Check current quiet setting
bid_get_quiet()
```

bid_ingest_telemetry *Ingest telemetry data and identify UX friction points*

Description

This function ingests telemetry data from multiple sources and automatically identifies potential UX issues, translating them into BID framework Notice stages. It returns a hybrid object that is backward-compatible as a list of Notice stages while also providing enhanced functionality with tidy tibble access and flags extraction.

Supported telemetry sources:

- shiny.telemetry (SQLite or JSON)
- Shiny native OpenTelemetry (Shiny >= 1.12.0, OTLP JSON or SQLite)
- DBI database connections

Format is automatically detected based on file structure and content.

OpenTelemetry Support: For Shiny >= 1.12.0 applications using native OpenTelemetry, pass the path to OTLP JSON exports or OTEL-formatted SQLite databases. Spans are automatically converted to events for analysis. See `vignette("otel-integration")` for complete setup guide.

Note: For Quarto dashboards, shiny.telemetry only works when using `server: shiny` in the Quarto YAML. Static Quarto dashboards and OJS-based dashboards do not support shiny.telemetry. Consider alternative analytics solutions (e.g., Plausible) for static dashboard usage tracking.

Usage

```
bid_ingest_telemetry(
  source,
  format = NULL,
  events_table = NULL,
  table_name = NULL,
  thresholds = list()
)
```

Arguments

source	Either a file path to telemetry data or a DBI connection object. Supports: • SQLite databases (shiny.telemetry or OTEL format) • JSON files (shiny.telemetry logs or OTLP JSON exports) • DBI connections to databases with event or span tables When a connection is provided, it will not be closed by this function.
format	Optional format specification ("sqlite", "json", "otlp_json", "otel_sqlite"). If NULL (default), auto-detected from file extension and structure. OTLP formats are automatically detected when file contains OpenTelemetry span data.
events_table	Optional data.frame specifying custom events table when reading from SQLite. Must have columns: event_id, timestamp, event_type, user_id. If NULL, auto-detects standard table names (event_data, events). Cannot be used with `table_name`.
table_name	Optional character string specifying the table name to read from the database. If NULL (default), auto-detects standard table names (event_data, events). Cannot be used with `events_table`.
thresholds	Optional list of threshold parameters: - unused_input_threshold: percentage of sessions below which input is considered unused (default: 0.05) - delay_threshold_secs: seconds of delay considered problematic (default: 30) - error_rate_threshold: percentage of sessions with errors considered problematic (default: 0.1) - navigation_threshold: percentage of sessions visiting a page below which it's considered underused (default: 0.2) - rapid_change_window: seconds within which multiple changes indicate confusion (default: 10) - rapid_change_count: number of changes within window to flag as confusion (default: 5)

Value

A hybrid object of class c("bid_issues", "list") containing bid_stage objects for each identified issue in the "Notice" stage. The object includes:

Legacy list	Named list of bid_stage objects (e.g., "unused_input_region", "delayed_interaction")
issues_tbl	Attached tidy tibble with issue metadata
flags	Global telemetry flags as named list
created_at	Timestamp when object was created

```
Use as_tibble() to access the tidy issues data, bid_flags() to extract flags,
 and legacy list access for backward compatibility.
```

Examples

```
## Not run:
# Works with shiny.telemetry SQLite
issues <- bid_ingest_telemetry("telemetry.sqlite")

# Works with Shiny OpenTelemetry (1.12+)
issues <- bid_ingest_telemetry("otel_spans.json")

# Use sensitivity presets for easier configuration
strict_issues <- bid_ingest_telemetry(
  "telemetry.sqlite",
  thresholds = bid_telemetry_presets("strict")
)

# Analyze JSON log with custom thresholds
issues <- bid_ingest_telemetry(
  "telemetry.log",
  format = "json",
  thresholds = list(
    unused_input_threshold = 0.1,
    delay_threshold_secs = 60
  )
)

# Use a DBI connection object directly
con <- DBI::dbConnect(RSQLite::SQLite(), "telemetry.sqlite")
issues <- bid_ingest_telemetry(con)
# Connection remains open for further use
DBI::dbDisconnect(con)

# Specify custom table name
issues <- bid_ingest_telemetry(
  "telemetry.sqlite",
  table_name = "my_custom_events"
)

# Same analysis workflow for both shiny.telemetry and OTEL
if (length(issues) > 0) {
  # Take first issue and continue with BID process
  interpret_result <- bid_interpret(
    previous_stage = issues[[1]],
    central_question = "How can we improve user engagement?"
  )
}

## End(Not run)
```

`bid_interpret` *Document User Interpretation Stage in BID Framework*

Description

This function documents the interpretation of user needs, capturing the central question and the data storytelling narrative. It represents stage 1 in the BID framework.

Usage

```
bid_interpret(
  previous_stage = NULL,
  central_question,
  data_story = NULL,
  user_personas = NULL,
  quiet = NULL
)
```

Arguments

previous_stage	Optional tibble or list output from an earlier BID stage function. Since Interpret is the first stage in the BID framework, this is typically NULL but can accept previous stage output in some iteration scenarios.
central_question	Required. A character string representing the main question to be answered. If NULL, will be suggested based on previous stage information.
data_story	A list containing elements such as hook, context, tension, resolution, and optionally audience, metrics, and visual_approach. If NULL, elements will be suggested based on previous stage.
user_personas	Optional list of user personas to consider in the design.
quiet	Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).

Value

A tibble containing the documented information for the "Interpret" stage.

Examples

```
# Recommended: use new_data_story() with flat API
interpret_result <- bid_interpret(
  central_question = "What drives the decline in user engagement?",
  data_story = new_data_story(
    hook = "Declining trend in engagement",
    context = "Previous high engagement levels",
    tension = "Unexpected drop",
    resolution = "Investigate new UI changes"
  )
)

# With user personas (using data.frame)
interpret_personas <- bid_interpret(
  central_question = "How can we improve data discovery?",
```

```
  data_story = new_data_story(
    hook = "Users are missing key insights",
    context = "Critical data is available but overlooked",
    tension = "Time-sensitive decisions are delayed",
    resolution = "Highlight key metrics more effectively",
    audience = "Data analysts and executives"
  ),
  user_personas = data.frame(
    name = c("Sara, Data Analyst", "Marcus, Executive"),
    goals = c(
      "Needs to quickly find patterns in data",
      "Wants high-level insights at a glance"
    ),
    pain_points = c(
      "Gets overwhelmed by too many visualizations",
      "Limited time to analyze detailed reports"
    ),
    technical_level = c("advanced", "beginner"),
    stringsAsFactors = FALSE
  )
)

summary(interpret_personas)
```

`bid_notice` *Document User Notice Stage in BID Framework*

Description

This function documents the observation and problem identification stage. It represents stage 2 in the BID framework and now returns a structured bid_stage object with enhanced metadata and external mapping support.

Usage

```
bid_notice(
  previous_stage,
  problem,
  theory = NULL,
  evidence = NULL,
  quiet = NULL,
  ...
)
```

Arguments

`previous_stage`	A tibble or list output from the previous BID stage function (typically bid_interpret).
`problem`	A character string describing the observed user problem.

theory	A character string describing the behavioral theory that might explain the problem. If NULL, will be auto-suggested using external theory mappings.
evidence	A character string describing evidence supporting the problem.
quiet	Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).
...	Additional parameters. Deprecated parameters (e.g., 'target_audience') will generate warnings if provided.

Value

A bid_stage object containing the documented information for the "Notice" stage with enhanced metadata and validation.

Examples

```
interpret_result <- bid_interpret(
  central_question = "How can we improve user task completion?",
  data_story = new_data_story(
    hook = "Users are struggling with complex interfaces",
    context = "Complex interfaces reducing completion",
    resolution = "Simplify key interactions"
  )
)

# Auto-suggested theory
bid_notice(
  previous_stage = interpret_result,
  problem = "Users struggling with complex dropdowns and too many options",
  evidence = "User testing shows 65% abandonment rate on filter selection"
)

# With explicit theory
notice_result <- bid_notice(
  previous_stage = interpret_result,
  problem = "Mobile interface is difficult to navigate",
  theory = "Fitts's Law",
  evidence = "Mobile users report frustration with small touch targets"
)

summary(notice_result)
```

---

bid_notices *Create multiple Notice stages from telemetry issues*

Description

Bridge function that converts multiple telemetry issues into Notice stages. Provides filtering and limiting options for managing large issue sets.

Usage

```
bid_notices(issues, filter = NULL, previous_stage = NULL, max_issues = 5, ...)
```

Arguments

issues	A tibble from bid_telemetry() output
filter	Optional filter expression for subsetting issues (e.g., severity == "critical")
previous_stage	Optional previous BID stage (typically from bid_interpret)
max_issues	Maximum number of issues to convert (default: 5)
...	Additional arguments passed to bid_notice_issue()

Value

A named list of bid_stage objects in the Notice stage

Examples

```
## Not run:
issues <- bid_telemetry("data.sqlite")
interpret <- bid_interpret("How can we reduce user friction?")

# Convert all critical issues
notices <- bid_notices(issues, filter = severity == "critical", interpret)

# Convert top 3 issues by impact
top_issues <- issues[order(-issues$impact_rate), ][1:3, ]
notices <- bid_notices(top_issues, previous_stage = interpret)

## End(Not run)
```

---

bid_notice_issue *Create Notice stage from individual telemetry issue*

Description

Bridge function that converts a single telemetry issue row into a BID Notice stage. This allows seamless integration between telemetry analysis and the BID framework.

Usage

```
bid_notice_issue(issue, previous_stage = NULL, override = list())
```

Arguments

issue	A single row from bid_telemetry() output or issues tibble
previous_stage	Optional previous BID stage (typically from bid_interpret)
override	List of values to override from the issue (problem, evidence, theory)

Value

A bid_stage object in the Notice stage

Examples

```
## Not run:
issues <- bid_telemetry("data.sqlite")
interpret <- bid_interpret("How can we reduce user friction?")

# Convert first issue to Notice stage
notice <- bid_notice_issue(issues[1, ], previous_stage = interpret)

# Override problem description
notice <- bid_notice_issue(
  issues[1, ],
  previous_stage = interpret,
  override = list(problem = "Custom problem description")
)

## End(Not run)
```

---

bid_pipeline *Create pipeline of Notice stages from top telemetry issues (sugar)*

Description

Convenience function that creates a pipeline of Notice stages from the highest priority telemetry issues. Useful for systematic issue resolution workflows.

Usage

```
bid_pipeline(issues, previous_stage, max = 3, ...)
```

Arguments

issues	A tibble from bid_telemetry() output
previous_stage	Previous BID stage (typically from bid_interpret)
max	Maximum number of issues to include in pipeline (default: 3)
...	Additional arguments passed to bid_notices()

Value

A named list of bid_stage objects in the Notice stage

Examples

```
## Not run:
issues <- bid_telemetry("data.sqlite")
interpret <- bid_interpret("How can we systematically improve UX?")

# Create pipeline for top 3 issues
notice_pipeline <- bid_pipeline(issues, interpret, max = 3)

# Continue with first issue in pipeline
anticipate <- bid_anticipate(previous_stage = notice_pipeline[[1]])

## End(Not run)
```

`bid_quick_suggest` *Quick UX Suggestions for R Dashboard Developers*

Description

Provides a streamlined, single-step workflow for R dashboard developers who need quick UX suggestions without going through the full 5-stage BID framework. This function internally leverages the BID framework stages but presents results in a simple, actionable format. Works with both Shiny applications and Quarto dashboards.

Unlike the full BID workflow (Interpret -> Notice -> Anticipate -> Structure -> Validate), this function provides immediate suggestions based on a problem description. Use this for rapid prototyping or when you need quick guidance. For comprehensive UX redesign projects, use the full BID workflow.

Usage

```
bid_quick_suggest(
  problem,
  context = NULL,
  package = NULL,
  limit = 10,
  min_score = 0.7,
  quiet = NULL
)
```

Arguments

`problem`	Required. A character string describing the UX problem. Examples: "Users can't find the download button", "Information overload on dashboard", "Mobile interface is hard to navigate".
`context`	Optional. Additional context about the application or users. This helps refine suggestions to your specific situation.

package	Optional. Filter suggestions to specific package ("bslib", "shiny", "reactable", "DT", "plotly", "leaflet", etc.). If NULL, returns suggestions for all relevant packages. Note: bslib, plotly, DT, reactable, and leaflet components work in both Shiny apps and Quarto dashboards.
limit	Optional. Maximum number of suggestions to return (default: 10). Set to Inf to return all suggestions.
min_score	Optional. Minimum relevance score 0-1 (default: 0.7). Higher values return only the most relevant suggestions.
quiet	Optional. Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).

Details

How it works:

The function analyzes your problem description using keyword matching and semantic analysis to:

1. Identify relevant UX concepts (cognitive load, navigation, visual hierarchy, etc.)
2. Detect appropriate layout patterns (grid, card, breathable, etc.)
3. Generate ranked suggestions with specific component recommendations
4. Filter and sort by relevance score

Problem Analysis Keywords:

- "overload", "overwhelm", "too many" -> Cognitive Load Theory
- "find", "search", "navigate" -> Information Scent
- "cluttered", "messy", "disorganized" -> Visual Hierarchy
- "mobile", "touch", "responsive" -> Fitts's Law
- "confusing", "unclear", "complex" -> Progressive Disclosure

When to use this vs full BID workflow:

- Use `bid_quick_suggest()`: Quick fixes, prototyping, single issues
- Use full workflow: Comprehensive redesigns, complex projects, team collaboration

Quarto Dashboard Compatibility: Component suggestions include both Shiny-specific (shiny::) and framework-agnostic components. For static Quarto dashboards or OJS-based interactivity, focus on bslib, DT, plotly, reactable, and leaflet suggestions. Shiny-prefixed components require `server: shiny` in Quarto dashboards or a traditional Shiny app.

Value

A tibble with columns:

title	Brief actionable description of the suggestion
details	Specific implementation guidance
components	R dashboard component recommendations (character vector). Components prefixed with 'shiny::' require Shiny runtime; bslib, DT, plotly, reactable, and leaflet components work in both Shiny and Quarto dashboards.

concept	UX concept the suggestion is based on
score	Relevance score (0-1, higher is more relevant)
difficulty	Implementation difficulty (easy/moderate/advanced)
rationale	1-2 sentence explanation of why this helps

Examples

```
# Basic usage
suggestions <- bid_quick_suggest(
  problem = "Users can't find the download button"
)
print(suggestions)

# With additional context
suggestions <- bid_quick_suggest(
  problem = "Dashboard has too many charts and metrics",
  context = "Financial analysts need quick insights but get overwhelmed",
  limit = 5
)

# Filter to specific package
bslib_suggestions <- bid_quick_suggest(
  problem = "Mobile interface is hard to use",
  package = "bslib",
  min_score = 0.8
)

# Navigation issues
nav_suggestions <- bid_quick_suggest(
  problem = "Users get lost in multi-tab interface",
  context = "Application has 10+ tabs with nested content"
)

# Information overload
overload_suggestions <- bid_quick_suggest(
  problem = "Too many filters and options on the sidebar",
  context = "Beginners find the interface overwhelming"
)
```

---

bid_report *Generate BID Framework Report*

Description

Creates a comprehensive report from a completed BID framework process. This report summarizes all stages and provides recommendations for implementation. Reports include component suggestions that work with both Shiny applications and Quarto dashboards (shiny-prefixed components (i.e., `shiny::`) require Shiny runtime).

Usage

```
bid_report(
  validate_stage,
  format = c("text", "html", "markdown"),
  include_diagrams = TRUE
)
```

Arguments

validate_stage	A tibble output from bid_validate().
format	Output format: "text", "html", or "markdown"
include_diagrams	Logical, whether to include ASCII diagrams in the report (default: TRUE)

Value

A formatted report summarizing the entire BID process

Examples

```
if (interactive()) {
  # After completing all 5 stages
  validation_result <- bid_validate(...)

  # Generate a text report
  bid_report(validation_result)

  # Generate an HTML report
  bid_report(validation_result, format = "html")

  # Generate a markdown report without diagrams
  bid_report(
    validation_result,
    format = "markdown",
    include_diagrams = FALSE
  )
}
```

---

bid_result *Constructor for BID result collection objects*

Description

Constructor for BID result collection objects

Usage

```
bid_result(stages)
```

Arguments

stages	List of bid_stage objects

Value

Object of class 'bid_result'

---

bid_set_quiet *Set global quiet mode for bidux functions*

Description

Convenience function to set the global quiet option for all bidux functions. When quiet mode is enabled, most informational messages are suppressed.

Usage

```
bid_set_quiet(quiet = TRUE)
```

Arguments

quiet	Logical indicating whether to enable quiet mode. When TRUE, most bidux messages are suppressed.

Value

The previous value of the quiet option (invisibly)

Examples

```
# Enable quiet mode
bid_set_quiet(TRUE)

# Disable quiet mode
bid_set_quiet(FALSE)
```

bid_stage *Constructor for BID stage objects*

Description

Constructor for BID stage objects

Usage

```
bid_stage(stage, data, metadata = list())
```

Arguments

stage	Character string indicating the stage name
data	Tibble containing the stage data
metadata	List containing additional metadata

Value

Object of class 'bid_stage'

bid_structure *Document Dashboard Structure Stage in BID Framework*

Description

This function documents the structure of the dashboard and generates ranked, concept-grouped actionable UI/UX suggestions. Returns structured recommendations with specific component pointers and implementation rationales.

Usage

```
bid_structure(
  previous_stage,
  concepts = NULL,
  telemetry_flags = NULL,
  quiet = NULL,
  ...
)
```

Arguments

previous_stage	A tibble or list output from an earlier BID stage function.
concepts	A character vector of additional BID concepts to include. Concepts can be provided in natural language (e.g., "Principle of Proximity") or with underscores (e.g., "principle_of_proximity"). The function uses fuzzy matching to identify the concepts. If NULL, will detect relevant concepts from previous stages automatically.
telemetry_flags	Optional named list of telemetry flags from bid_flags(). Used to adjust suggestion scoring based on observed user behavior patterns.
quiet	Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).
...	Additional parameters (reserved for future use).

Details

Suggestion Engine: Generates ranked, actionable recommendations grouped by UX concepts. Each suggestion includes specific R dashboard components (Shiny, bslib, DT, plotly, etc.), implementation details, and rationale. Suggestions are scored based on relevance and contextual factors. Component suggestions work with both Shiny applications and Quarto dashboards, with shiny-prefixed components (i.e., `shiny::`) requiring Shiny runtime.

Value

A bid_stage object containing:

stage	"Structure"
suggestions	List of concept groups with ranked suggestions (nested format)
suggestions_tbl	Flattened tibble with all suggestions, includes columns: concept, title, details, components, rationale, score, difficulty, category
concepts	Comma-separated string of all concepts used

Examples

```
notice_result <- bid_interpret(
  central_question = "How can we simplify data presentation?",
  data_story = new_data_story(
    hook = "Data is too complex",
    context = "Overloaded with charts",
    tension = "Confusing layout",
    resolution = "Introduce clear grouping"
  )
) |>
  bid_notice(
    problem = "Users struggle with information overload",
    evidence = "Survey results indicate delays"
  )
```

```
# Generate concept-grouped suggestions
structure_result <- bid_structure(previous_stage = notice_result)
print(structure_result$suggestions) # Ranked suggestions by concept (nested)

# Access flattened tibble format for easier manipulation
suggestions_flat <- structure_result$suggestions_tbl[[1]]
print(suggestions_flat)

# Filter by difficulty
easy_suggestions <- suggestions_flat[suggestions_flat$difficulty == "Easy", ]

# Filter by category
layout_suggestions <- suggestions_flat[suggestions_flat$category == "Layout", ]

summary(structure_result)
```

`bid_suggest_analytics` *Suggest alternative analytics solutions for static dashboards*

Description

Provides recommendations for analytics and telemetry solutions suitable for static Quarto dashboards, where Shiny-based telemetry (shiny.telemetry or OpenTelemetry) is not available. This function helps you choose the right analytics tool based on your needs and constraints.

Important: shiny.telemetry and Shiny OpenTelemetry only work with `server: shiny` in Quarto YAML. For static Quarto dashboards (including OJS-based dashboards), you need alternative web analytics solutions.

Usage

```
bid_suggest_analytics(
  dashboard_type = c("static", "ojs", "python"),
  privacy_preference = c("gdpr_compliant", "privacy_focused", "standard"),
  budget = c("flexible", "free", "low"),
  self_hosted = FALSE
)
```

Arguments

`dashboard_type` Character string specifying the type of dashboard:

- **static** Static HTML Quarto dashboard (default)
- **ojs** Quarto dashboard using Observable JS
- **python** Static dashboard with Python/Jupyter

`privacy_preference`

Character string indicating privacy requirements:

gdpr_compliant Prioritize GDPR-compliant solutions (default)
privacy_focused Emphasize user privacy and no tracking
standard Standard analytics with typical tracking

`budget` Character string indicating budget constraints:
free Only free/open-source solutions
low Low-cost solutions (< $10/month)
flexible Any cost tier (default)

`self_hosted` Logical indicating whether self-hosted solutions are preferred (default: FALSE)

Value

A data frame with recommended analytics solutions, including:

`solution` Name of the analytics platform
`type` Type of solution (privacy-focused, traditional, open-source)
`cost` Cost tier (free, paid, freemium)
`self_hosted` Whether self-hosting is available
`gdpr_compliant` Whether the solution is GDPR compliant
`integration_method` How to integrate (script tag, API, etc.)
`key_features` Main features for UX analysis
`bidux_compatibility` How well it works with BID framework
`docs_url` Link to integration documentation

Integration Patterns

For Static Quarto Dashboards:

1. **Event Tracking** - Track user interactions with custom events:
   - Button clicks, filter changes, tab switches
   - Use JavaScript event listeners in Quarto
   - Send events to analytics platform via API
2. **Session Analysis** - Monitor user sessions:
   - Page views, time on page, bounce rate
   - User flow through dashboard sections
   - Identify drop-off points
3. **Custom Dimensions** - Track dashboard-specific metrics:
   - Selected filters, date ranges, visualization types
   - User cohorts, roles, or departments
   - Dashboard version or configuration

Example Integration (Plausible Analytics):

Add to your Quarto dashboard header:

```
<script defer data-domain="yourdomain.com"
  src="https://plausible.io/js/script.tagged-events.js"></script>
```

Track custom events in your dashboard JavaScript:

```
// Track filter change
document.getElementById('regionFilter').addEventListener('change', function(e) {
 plausible('Filter Changed', {props: {filter: 'region', value: e.target.value}});
});

// Track visualization interaction
plotElement.on('plotly_click', function(data) {
 plausible('Chart Interaction', {props: {chart: 'sales_plot', action: 'click'}});
});
```

Analyzing Results with BID Framework:

While these analytics tools won't automatically integrate with `bid_ingest_telemetry()`, you can still apply BID framework principles:

1. **Notice** - Export analytics data, identify friction points manually
2. **Interpret** - Use `bid_interpret()` with insights from analytics
3. **Anticipate** - Apply `bid_anticipate()` to plan improvements
4. **Structure** - Design improvements with `bid_structure()`
5. **Validate** - Measure impact with before/after analytics comparison

Examples

```
# Get recommendations for static Quarto dashboard with GDPR compliance
suggestions <- bid_suggest_analytics(
  dashboard_type = "static",
  privacy_preference = "gdpr_compliant"
)
print(suggestions)

# Find free, privacy-focused solutions for OJS dashboard
privacy_options <- bid_suggest_analytics(
  dashboard_type = "ojs",
  privacy_preference = "privacy_focused",
  budget = "free"
)

# Get self-hosted options
self_hosted <- bid_suggest_analytics(
  dashboard_type = "static",
  self_hosted = TRUE
)

# View top recommendation
top_choice <- suggestions[1, ]
```

```
cat(sprintf("Recommended: %s\n", top_choice$solution))
cat(sprintf("Integration: %s\n", top_choice$integration_method))
cat(sprintf("Docs: %s\n", top_choice$docs_url))
```

`bid_suggest_components` *Suggest UI Components Based on BID Framework Analysis*

Description

This function analyzes the results from BID framework stages and suggests appropriate UI components from popular R dashboard packages like shiny, bslib, DT, plotly, reactable, and htmlwidgets. The suggestions are based on the design principles and user needs identified in the BID process. Components work with both Shiny applications and Quarto dashboards (shiny-prefixed components require Shiny runtime).

Usage

```
bid_suggest_components(bid_stage, package = NULL)
```

Arguments

`bid_stage`	A tibble output from any BID framework stage function
`package`	Optional character string specifying which package to focus suggestions on. Options include "shiny", "bslib", "DT", "plotly", "reactable", "htmlwidgets". If NULL, suggestions from all packages are provided.

Value

A tibble containing component suggestions with relevance scores

Examples

```
if (interactive()) {
  # After completing BID stages
  notice_result <- bid_notice(
    problem = "Users struggle with complex data",
    theory = "Cognitive Load Theory"
  )

  # Get all component suggestions
  bid_suggest_components(notice_result)

  # Get only bslib suggestions
  bid_suggest_components(notice_result, package = "bslib")

  # Get shiny-specific suggestions
  bid_suggest_components(notice_result, package = "shiny")
```

```
}
```

`bid_telemetry` *Concise telemetry analysis with tidy output*

Description

Preferred modern interface for telemetry analysis. Returns a clean tibble of identified issues without the legacy list structure. Use this function for new workflows that don't need backward compatibility.

OpenTelemetry Support: For Shiny >= 1.12.0 applications using native OpenTelemetry, pass the path to OTLP JSON exports or OTEL-formatted SQLite databases. Format is auto-detected. See `vignette("otel-integration")` for complete setup guide.

Usage

```
bid_telemetry(
  source,
  format = NULL,
  events_table = NULL,
  table_name = NULL,
  thresholds = list()
)
```

Arguments

`source`	Either a file path to telemetry data or a DBI connection object. Supports: • SQLite databases (shiny.telemetry or OTEL format) • JSON files (shiny.telemetry logs or OTLP JSON exports) • DBI connections to databases with event or span tables When a connection is provided, it will not be closed by this function.
`format`	Optional format specification ("sqlite", "json", "otlp_json", "otel_sqlite"). If NULL (default), auto-detected from file extension and structure. OTLP formats are automatically detected when file contains OpenTelemetry span data.
`events_table`	Optional data.frame specifying custom events table when reading from SQLite. Must have columns: event_id, timestamp, event_type, user_id. If NULL, auto-detects standard table names (event_data, events). Cannot be used with `table_name`.
`table_name`	Optional character string specifying the table name to read from the database. If NULL (default), auto-detects standard table names (event_data, events). Cannot be used with `events_table`.
`thresholds`	Optional list of threshold parameters: - unused_input_threshold: percentage of sessions below which input is considered unused (default: 0.05) - delay_threshold_secs: seconds of delay considered problematic (default: 30) - error_rate_threshold:

percentage of sessions with errors considered problematic (default: 0.1) - navigation_threshold: percentage of sessions visiting a page below which it's considered underused (default: 0.2) - rapid_change_window: seconds within which multiple changes indicate confusion (default: 10) - rapid_change_count: number of changes within window to flag as confusion (default: 5)

Value

A tibble of class "bid_issues_tbl" with structured issue metadata

Examples

```
## Not run:
# Works with shiny.telemetry
issues <- bid_telemetry("telemetry.sqlite")

# Works with Shiny OpenTelemetry (1.12+)
issues <- bid_telemetry("otel_spans.json")

# Same analysis workflow for both
high_priority <- issues[issues$severity %in% c("critical", "high"), ]

# Use DBI connection directly
con <- DBI::dbConnect(RSQLite::SQLite(), "telemetry.sqlite")
issues <- bid_telemetry(con, table_name = "my_events")
DBI::dbDisconnect(con)

# Use with bridges for BID workflow
top_issue <- issues[1, ]
notice <- bid_notice_issue(top_issue, previous_stage = interpret_stage)

## End(Not run)
```

---

bid_telemetry_presets *Get predefined telemetry sensitivity presets*

Description

Returns predefined threshold configurations for telemetry analysis with different sensitivity levels. Use these presets with `bid_ingest_telemetry()` or `bid_telemetry()` to easily adjust how aggressively the analysis identifies UX friction points.

OpenTelemetry Compatibility: These presets work with both shiny.telemetry event data and Shiny 1.12+ OpenTelemetry span data. When using OTEL data, spans are automatically converted to events for analysis.

Usage

```
bid_telemetry_presets(preset = c("moderate", "strict", "relaxed"))
```

Arguments

preset	Character string specifying the sensitivity level:
	strict Detects even minor issues - use for critical applications or new dashboards
	moderate Balanced default - appropriate for most applications (default)
	relaxed Only detects major issues - use for mature, stable dashboards

Value

Named list of threshold parameters suitable for passing to `bid_ingest_telemetry()` or `bid_telemetry()` thresholds parameter.

Examples

```
# Get strict sensitivity thresholds
strict_thresholds <- bid_telemetry_presets("strict")

# Use with telemetry analysis (works with both shiny.telemetry and OTEL)
## Not run:
# Works with shiny.telemetry
issues <- bid_telemetry(
  "telemetry.sqlite",
  thresholds = bid_telemetry_presets("strict")
)

# Works with Shiny OpenTelemetry (1.12+)
issues <- bid_telemetry(
  "otel_spans.json",
  thresholds = bid_telemetry_presets("strict")
)

## End(Not run)

# Compare different presets
moderate <- bid_telemetry_presets("moderate")
relaxed <- bid_telemetry_presets("relaxed")
```

---

bid_validate *Document User Validation Stage in BID Framework*

Description

This function documents the validation stage, where the user tests and refines the dashboard. It represents stage 5 in the BID framework.

Usage

```
bid_validate(
  previous_stage,
  summary_panel = NULL,
  collaboration = NULL,
  next_steps = NULL,
  include_exp_design = TRUE,
  include_telemetry = TRUE,
  telemetry_refs = NULL,
  include_empower_tools = TRUE,
  quiet = NULL
)
```

Arguments

previous_stage	A tibble or list output from an earlier BID stage function.
summary_panel	A character string describing the final summary panel or key insight presentation.
collaboration	A character string describing how the dashboard enables collaboration and sharing.
next_steps	A character vector or string describing recommended next steps for implementation and iteration.
include_exp_design	Logical indicating whether to include experiment design suggestions. Default is TRUE.
include_telemetry	Logical indicating whether to include telemetry tracking and monitoring suggestions. Default is TRUE.
telemetry_refs	Optional character vector or named list specifying specific telemetry reference points to include in validation steps. If provided, these will be integrated into the telemetry tracking recommendations with provenance information.
include_empower_tools	Logical indicating whether to include context-aware empowerment tool suggestions. Default is TRUE.
quiet	Logical indicating whether to suppress informational messages. If NULL, uses getOption("bidux.quiet", FALSE).

Value

A tibble containing the documented information for the "Validate" stage.

Examples

```
validate_result <- bid_interpret(
  central_question = "How can we improve delivery efficiency?",
  data_story = new_data_story(
    hook = "Too many delays",
```

```
    context = "Excessive shipments",
    tension = "User frustration",
    resolution = "Increase delivery channels"
  )
) |>
  bid_notice(
    problem  = "Issue with dropdown menus",
    evidence = "User testing indicated delays"
  ) |>
  bid_anticipate(
    bias_mitigations = list(
      anchoring = "Provide reference points",
      framing = "Use gain-framed messaging"
    )
  ) |>
  bid_structure() |>
  bid_validate(
    include_exp_design = FALSE,
    include_telemetry = TRUE,
    include_empower_tools = TRUE
  )

summary(validate_result)
```

`bid_with_quiet` *Temporarily suppress bidux messages*

Description

Execute code with bidux messages temporarily suppressed.

Usage

```
bid_with_quiet(code)
```

Arguments

code	Code to execute with messages suppressed

Value

The result of evaluating code

Examples

```
# Run analysis quietly without changing global setting
result <- bid_with_quiet({
  bid_interpret(
    central_question = "How can we improve user engagement?",
```

```
    data_story = new_data_story(
      hook = "Users are leaving",
      context = "User engagement declining",
      resolution = "Fix issues"
    )
  )
})
```

`convert_otel_spans_to_events` *Convert OTLP spans to bidux event schema*

Description

Converts OpenTelemetry Protocol (OTLP) span data to the bidux telemetry event schema. This enables transparent compatibility with existing friction detection algorithms. This function is called automatically by `bid_telemetry()` and `bid_ingest_telemetry()` when OTLP data is detected - you rarely need to call it directly.

Automatic Format Detection: When you pass OTLP JSON or SQLite to `bid_telemetry()`, this conversion happens automatically. The same UX friction detection algorithms work seamlessly on both shiny.telemetry events and OpenTelemetry spans.

Span to Event Mapping:

- session_start -> login
- output -> output
- reactive, observe -> input
- reactive_update -> synthetic timing events
- Error span events -> error

Usage

```
convert_otel_spans_to_events(spans_df)
```

Arguments

`spans_df`	Data frame of OTLP spans with columns: • name: span name (e.g., "session_start", "output:plot1") • startTimeUnixNano: start timestamp in Unix nanoseconds • endTimeUnixNano: end timestamp in Unix nanoseconds • attributes: list column with span attributes • events: list column with span events (for errors)

Value

Tibble with bidux event schema columns:

- timestamp: POSIXct event timestamp
- session_id: character session identifier
- event_type: character event type (login, input, output, error)
- input_id: character input identifier (NA for non-input events)
- value: character/numeric value (NA for most otel spans)
- error_message: character error message (NA for non-error events)
- output_id: character output identifier (NA for non-output events)
- navigation_id: character navigation identifier (NA for otel spans)

See Also

- `bid_telemetry()` for high-level telemetry analysis (automatic format detection)
- `bid_ingest_telemetry()` for legacy telemetry workflows
- `vignette("otel-integration")` for complete OTEL setup guide

Examples

```
## Not run:
# Typically you don't need to call this directly - use bid_telemetry() instead:
issues <- bid_telemetry("otel_spans.json")

# Manual conversion (advanced use case):
# After reading otlp json file
spans <- read_otel_json("spans.json")
events <- convert_otel_spans_to_events(spans)

# Verify schema compatibility
names(events)
# [1] "timestamp" "session_id" "event_type" "input_id" "value" "error_message"
# [7] "output_id" "navigation_id"

# Now use standard friction detection
issues <- detect_telemetry_issues(events)

## End(Not run)
```

`extract_stage` *Extract specific stage from bid_result*

Description

Extract specific stage from bid_result

Usage

```
extract_stage(workflow, stage)
```

Arguments

`workflow`	A bid_result object
`stage`	Character string with stage name

Value

A bid_stage object or NULL if not found

`get_accessibility_recommendations` *Get accessibility recommendations for a given context*

Description

Get accessibility recommendations for a given context

Usage

```
get_accessibility_recommendations(context = "", guidelines = NULL)
```

Arguments

`context`	Character string describing the interface context
`guidelines`	Optional custom accessibility guidelines

Value

Character vector of relevant accessibility recommendations

get_concept_bias_mappings *Get bias mitigation strategies for concepts*

Description

Get bias mitigation strategies for concepts

Usage

```
get_concept_bias_mappings(concepts, mappings = NULL)
```

Arguments

concepts	Character vector of concept names
mappings	Optional custom concept-bias mappings

Value

Data frame with relevant bias mappings

get_layout_concepts *Get concepts recommended for a layout*

Description

Get concepts recommended for a layout

Usage

```
get_layout_concepts(layout, mappings = NULL)
```

Arguments

layout	Character string indicating layout type
mappings	Optional custom layout mappings

Value

Character vector of recommended concepts

get_metadata *Get metadata from bid_stage object*

Description

Get metadata from bid_stage object

Usage

```
get_metadata(x)
```

Arguments

x	A bid_stage object

Value

List with metadata

get_stage *Get stage name from bid_stage object*

Description

Get stage name from bid_stage object

Usage

```
get_stage(x)
```

Arguments

x	A bid_stage object

Value

Character string with stage name

is_bid_stage *Check if object is a bid_stage*

Description

Check if object is a bid_stage

Usage

```
is_bid_stage(x)
```

Arguments

x	Object to test

Value

Logical indicating if object is bid_stage

is_complete *Check if workflow is complete (has all 5 stages)*

Description

Check if workflow is complete (has all 5 stages)

Usage

```
is_complete(x)
```

Arguments

x	A bid_result object

Value

Logical indicating if workflow is complete

new_bias_mitigations *Create bias mitigations tibble*

Description

Creates a structured bias mitigations tibble for use in bidux functions. Replaces nested list structures with a validated data.frame structure.

Usage

```
new_bias_mitigations(mitigations_df)
```

Arguments

`mitigations_df` Data.frame with required columns: bias_type, mitigation_strategy, confidence_level

Value

A bid_bias_mitigations S3 object (inherits from data.frame)

Examples

```
mitigations <- new_bias_mitigations(data.frame(
  bias_type = c("confirmation_bias", "selection_bias"),
  mitigation_strategy = c("seek_disconfirming_evidence", "randomize_sample"),
  confidence_level = c(0.8, 0.7)
))
```

new_data_story *Create a data story object*

Description

Creates a structured data story object for use in bid_interpret() and other bidux functions. Uses a flat API with hook, context, tension, and resolution fields to structure your data narrative.

Usage

```
new_data_story(
  hook = NULL,
  context = NULL,
  tension = NULL,
  resolution = NULL,
  ...
)
```

Arguments

hook	Character string for the story hook (attention-grabbing opening)
context	Character string describing the data context
tension	Character string describing the problem or tension
resolution	Character string describing the resolution or next steps
...	Optional additional fields (audience, metrics, visual_approach, etc.)

Value

A bid_data_story S3 object

Examples

```
# Basic usage
story <- new_data_story(
  hook = "User engagement is declining",
  context = "Our dashboard usage has dropped 30% this quarter",
  tension = "We don't know if it's UX issues or changing user needs",
  resolution = "Analyze telemetry to identify friction points"
)

# With optional fields
story_detailed <- new_data_story(
  hook = "Revenue dashboards are underutilized",
  context = "Only 40% of sales team uses the new revenue dashboard",
  tension = "Critical metrics are being missed",
  resolution = "Redesign with behavioral science principles",
  audience = "Sales team",
  metrics = "adoption_rate, time_to_insight"
)
```

---

new_user_personas *Create user personas tibble*

Description

Creates a structured user personas tibble for use in bidux functions. Replaces nested list structures with a validated data.frame structure.

Usage

```
new_user_personas(personas_df)
```

Arguments

personas_df	Data.frame with required columns: name, goals, pain_points, technical_level

Value

A bid_user_personas S3 object (inherits from data.frame)

Examples

```
personas <- new_user_personas(data.frame(
  name = c("data analyst", "product manager"),
  goals = c("quick insights", "strategic overview"),
  pain_points = c("complex tools", "data delays"),
  technical_level = c("intermediate", "beginner")
))
```

---

print.bid_bias_mitigations *Print method for bias mitigations objects*

Description

Print method for bias mitigations objects

Usage

```
## S3 method for class 'bid_bias_mitigations'
print(x, ...)
```

Arguments

x	A bid_bias_mitigations object
...	Additional arguments passed to print.data.frame

---

print.bid_data_story *Print method for data story objects*

Description

Print method for data story objects

Usage

```
## S3 method for class 'bid_data_story'
print(x, ...)
```

Arguments

x	A bid_data_story object
...	Additional arguments (unused)

print.bid_issues *Print method for bid_issues objects*

Description

Displays a triage view of telemetry issues with severity-based prioritization and provides a reminder about legacy list access for backward compatibility.

Usage

```
## S3 method for class 'bid_issues'
print(x, ...)
```

Arguments

x	A bid_issues object from bid_ingest_telemetry()
...	Additional arguments (unused)

Value

Invisible x (for chaining)

print.bid_result *Print method for BID result objects*

Description

Print method for BID result objects

Usage

```
## S3 method for class 'bid_result'
print(x, ...)
```

Arguments

x	A bid_result object
...	Additional arguments

Value

Returns the input `bid_result` object invisibly (class: `c("bid_result", "list")`). The method is called for its side effects: printing a workflow overview to the console showing completion status, stage progression, and key information from each completed BID stage. The invisible return supports method chaining while emphasizing the console summary output.

print.bid_stage *Print method for BID stage objects*

Description

Print method for BID stage objects

Usage

```
## S3 method for class 'bid_stage'
print(x, ...)
```

Arguments

x	A bid_stage object
...	Additional arguments

Value

Returns the input `bid_stage` object invisibly (class: `c("bid_stage", "tbl_df", "tbl", "data.frame")`). The method is called for its side effects: printing a formatted summary of the BID stage to the console, including stage progress, key stage-specific information, and usage suggestions. The invisible return allows for method chaining while maintaining the primary purpose of console output.

print.bid_user_personas *Print method for user personas objects*

Description

Print method for user personas objects

Usage

```
## S3 method for class 'bid_user_personas'
print(x, ...)
```

Arguments

x	A bid_user_personas object
...	Additional arguments passed to print.data.frame

suggest_theory_from_mappings *Suggest theory based on problem and evidence using mappings*

Description

Suggest theory based on problem and evidence using mappings

Usage

```
suggest_theory_from_mappings(problem, evidence = NULL, mappings = NULL)
```

Arguments

problem	Character string describing the problem
evidence	Optional character string with supporting evidence
mappings	Optional custom theory mappings

Value

Character string with suggested theory

summary.bid_result *Summary method for BID result objects*

Description

Summary method for BID result objects

Usage

```
## S3 method for class 'bid_result'
summary(object, ...)
```

Arguments

object	A bid_result object
...	Additional arguments

Value

Returns the input `bid_result` object invisibly (class: `c("bid_result", "list")`). The method is called for its side effects: printing a detailed workflow analysis to the console including completion statistics, duration metrics, and comprehensive stage-by-stage breakdowns with key data from each BID framework stage. The invisible return facilitates method chaining while focusing on comprehensive console reporting.

`summary.bid_stage` *Summary method for BID stage objects*

Description

Summary method for BID stage objects

Usage

```
## S3 method for class 'bid_stage'
summary(object, ...)
```

Arguments

`object`	A bid_stage object
`...`	Additional arguments

Value

Returns the input `bid_stage` object invisibly (class: `c("bid_stage", "tbl_df", "tbl", "data.frame")`). The method is called for its side effects: printing a comprehensive summary to the console including stage metadata, all non-empty data columns, and timestamp information. The invisible return enables method chaining while prioritizing the detailed console output display.

Index