Beads PureScript Port

Status: draft **Category**: plan **Created**: 2026-01-28 **Tags**: beads, issue-tracking, purescript, cli, graph

Motivation

Beads is a distributed, git-backed issue tracker designed for AI coding agents. It's reportedly "100% vibe coded" — we want to understand the architecture by rebuilding it with types.

Goals:

1. Understand the actual architecture beneath the Go implementation
2. Get a typed specification of the beads protocol
3. Have a version we trust to run on our codebase
4. Potentially extend with PSD3 visualization (dependency graphs)

Non-goals (initially):

- Feature parity with Go version
- Performance optimization
- Daemon/background sync

Core Types

Identity

```
-- Hash-based IDs to prevent merge collisions
newtype IssueId = IssueId String

derive instance Eq IssueId
derive instance Ord IssueId
derive newtype instance Show IssueId

-- Generate from UUID, take first N chars of hash
-- N grows as database grows (4 → 5 → 6)
generateId :: UUID -> Int -> IssueId
generateId uuid len = IssueId $ "bd-" <> take len (sha256hex uuid)

-- Hierarchical IDs for epics
-- bd-a3f8 (epic) → bd-a3f8.1 (task) → bd-a3f8.1.1 (subtask)
data HierarchicalId
  = RootId IssueId
  | ChildId IssueId Int
  | GrandchildId IssueId Int Int

parseId :: String -> Either String IssueId
```

Status & Classification

```
data Status
  = Open
  | InProgress
  | Blocked
  | Deferred
  | Closed
  | Tombstone  -- soft delete

derive instance Eq Status
derive instance Generic Status _
instance EncodeJson Status where ...
instance DecodeJson Status where ...

data Priority
  = P0  -- critical
  | P1  -- high
  | P2  -- medium (default)
  | P3  -- low
  | P4  -- backlog

data IssueType
  = Bug
  | Feature
  | Task
  | Epic
  | Chore
  | Molecule  -- template instance
  | Wisp      -- ephemeral

-- Priority is ordered: P0 < P1 < P2 < P3 < P4
derive instance Ord Priority
```

The Issue Record

```
type Issue =
  { id :: IssueId
  , title :: String
  , description :: Maybe String
  , status :: Status
  , priority :: Priority
  , issueType :: IssueType

  -- Assignment
  , assignee :: Maybe String
  , estimatedMinutes :: Maybe Int

  -- Timestamps
  , createdAt :: DateTime
```

```
  , createdBy :: Maybe String
  , updatedAt :: DateTime
  , closedAt :: Maybe DateTime
  , closeReason :: Maybe String

  -- Relations (denormalized for JSONL simplicity)
  , labels :: Array String
  , dependencies :: Array Dependency
  , comments :: Array Comment

  -- External integration
  , externalRef :: Maybe String  -- gh-123, jira-ABC

  -- Soft delete
  , deletedAt :: Maybe DateTime
  , deletedBy :: Maybe String
  , deleteReason :: Maybe String
  }

-- Smart constructor with defaults
mkIssue :: IssueId -> String -> Issue
mkIssue id title =
  { id
  , title
  , description: Nothing
  , status: Open
  , priority: P2
  , issueType: Task
  , assignee: Nothing
  , estimatedMinutes: Nothing
  , createdAt: unsafePerformEffect now  -- placeholder
  , createdBy: Nothing
  , updatedAt: unsafePerformEffect now
  , closedAt: Nothing
  , closeReason: Nothing
  , labels: []
  , dependencies: []
  , comments: []
  , externalRef: Nothing
  , deletedAt: Nothing
  , deletedBy: Nothing
  , deleteReason: Nothing
  }
```

Dependencies (The Graph)

```
data DependencyType
  = Blocks        -- X blocks Y: Y cannot start until X closes
  | ParentChild   -- X is parent of Y (epic/subtask)
  | Related       -- soft link, informational
  | DiscoveredFrom -- Y was discovered while working on X
```

```
type Dependency =
  { toId :: IssueId
  , depType :: DependencyType
  }

-- The dependency graph
type IssueGraph = Map IssueId Issue

-- Core graph queries
blockedBy :: IssueGraph -> IssueId -> Array IssueId
blockedBy graph id =
  case Map.lookup id graph of
    Nothing -> []
    Just issue ->
      issue.dependencies
        # filter (\d -> d.depType == Blocks)
        # map _.toId
        # filter (\bid -> isOpen (Map.lookup bid graph))

-- THE KEY FUNCTION: find ready issues
-- An issue is "ready" if:
-- 1. It's open (not closed, not blocked status)
-- 2. All its Blocks dependencies are closed
ready :: IssueGraph -> Array Issue
ready graph =
  graph
    # Map.values
    # Array.fromFoldable
    # filter isOpen
    # filter (hasNoOpenBlockers graph)

isOpen :: Maybe Issue -> Boolean
isOpen (Just i) = i.status == Open || i.status == InProgress
isOpen Nothing = false

hasNoOpenBlockers :: IssueGraph -> Issue -> Boolean
hasNoOpenBlockers graph issue =
  blockedBy graph issue.id
    # all (\bid -> not (isOpen (Map.lookup bid graph)))
```

Comments & Events

```
type Comment =
  { id :: String
  , author :: Maybe String
  , content :: String
  , createdAt :: DateTime
  }

data EventType
```

```
    = Created
    | Updated (Array String)  -- changed fields
    | StatusChanged Status Status
    | DependencyAdded Dependency
    | DependencyRemoved IssueId DependencyType
    | Commented String
    | Closed String  -- reason

  type Event =
    { id :: String
    , issueId :: IssueId
    , eventType :: EventType
    , timestamp :: DateTime
    , actor :: Maybe String
    }
```

Chemistry Types (Phase 2)

```
-- Work item phases as phantom types
data Proto   -- frozen template
data Mol     -- persistent, git-synced
data Wisp    -- ephemeral, local-only

-- Indexed by phase
newtype WorkItem phase = WorkItem Issue

-- Type-safe operations
pour :: WorkItem Proto -> Effect (WorkItem Mol)
wispCreate :: WorkItem Proto -> Effect (WorkItem Wisp)

close :: WorkItem Mol -> String -> Effect (WorkItem Mol)

squash :: WorkItem Wisp -> Effect (WorkItem Mol)  -- wisp → digest
burn :: WorkItem Wisp -> Effect Unit              -- wisp → gone

-- Proto/Mol can be closed normally
-- Wisp must be squashed or burned
-- This is enforced by types!
```

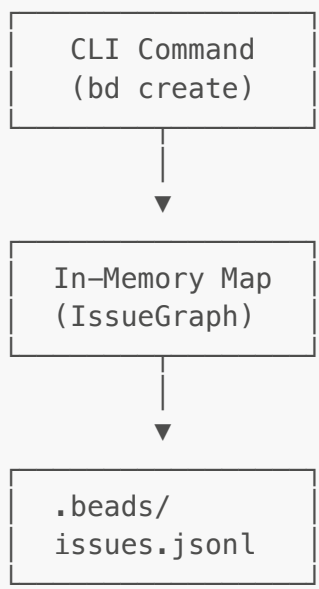
JSONL Layer

```
-- One issue per line
type JSONL = Array String

parseJSONL :: String -> Either JsonDecodeError (Array Issue)
parseJSONL content =
  content
```

```
    # String.split (Pattern "\n")
    # filter (not <<< String.null)
    # traverse decodeJson

serializeJSONL :: Array Issue -> String
serializeJSONL issues =
  issues
    # map encodeJson
    # map stringify
    # String.joinWith "\n"

-- File operations
readIssuesFile :: FilePath -> Aff (Either Error (Array Issue))
readIssuesFile path = do
  content <- readTextFile UTF8 path
  pure $ parseJSONL content

writeIssuesFile :: FilePath -> Array Issue -> Aff Unit
writeIssuesFile path issues =
  writeTextFile UTF8 path (serializeJSONL issues <> "\n")
```

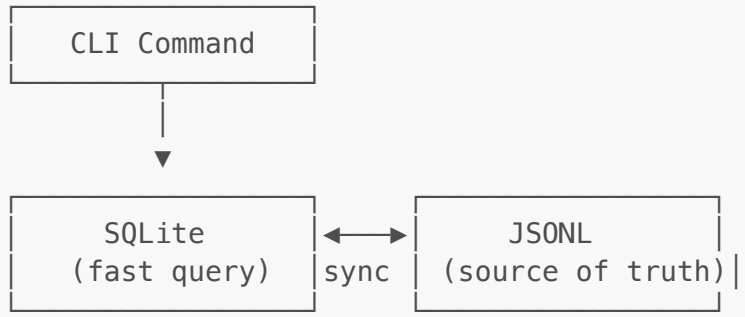
Storage Architecture

Phase 1: JSONL-only (MVP)



Simple: read entire file into memory, modify, write back. Good enough for hundreds of issues. Thousands might get slow.

Phase 2: SQLite Cache



Add SQLite for:

- Fast queries on large datasets
- Full-text search
- Complex dependency queries

Use `purescript-sqlite` or FFI to better-sqlite3.

CLI Design

Command Structure

```
bd <command> [options]

Commands:
  init                    Initialize .beads/ in current directory
  create <title>          Create new issue
  list [--status=open]    List issues
  show <id>               Show issue details
  update <id> [--field=value]  Update issue
  close <id> [--reason]   Close issue
  reopen <id>             Reopen closed issue

  dep add <from> <to> [--type=blocks]   Add dependency
  dep remove <from> <to>                Remove dependency
  dep show <id>                         Show dependency graph

  ready                   List issues ready to work on

  sync                    Export to JSONL (if using SQLite)
  import                  Import from JSONL (if using SQLite)
```

Implementation

```
-- Using purescript-optparse or similar
data Command
  = Init
```

```
  | Create { title :: String, priority :: Maybe Priority }
  | List { status :: Maybe Status, assignee :: Maybe String }
  | Show { id :: IssueId }
  | Update { id :: IssueId, updates :: Array FieldUpdate }
  | Close { id :: IssueId, reason :: Maybe String }
  | Reopen { id :: IssueId }
  | DepAdd { from :: IssueId, to :: IssueId, depType :: DependencyType }
  | DepRemove { from :: IssueId, to :: IssueId }
  | Ready

data FieldUpdate
  = SetTitle String
  | SetDescription String
  | SetPriority Priority
  | SetAssignee String
  | AddLabel String
  | RemoveLabel String

-- Main dispatch
runCommand :: Command -> Aff Unit
runCommand = case _ of
  Init -> initBeads
  Create opts -> createIssue opts
  List opts -> listIssues opts
  Ready -> listReady
  ...
```

Module Structure

```
Beads/
├── Core/
│   ├── Types.purs          -- Issue, Status, Priority, etc.
│   ├── Id.purs             -- IssueId generation, parsing
│   └── Graph.purs          -- IssueGraph, ready algorithm
│
├── Storage/
│   ├── JSONL.purs          -- Parse/serialize JSONL
│   ├── FileSystem.purs     -- Read/write .beads/
│   └── SQLite.purs         -- (Phase 2) Cache layer
│
├── Chemistry/              -- (Phase 2)
│   ├── Types.purs          -- Proto, Mol, Wisp phantom types
│   ├── Pour.purs           -- Template instantiation
│   └── Lifecycle.purs      -- Squash, burn
│
├── CLI/
│   ├── Parser.purs         -- Command line parsing
│   ├── Commands.purs       -- Command implementations
│   ├── Output.purs         -- Formatting (plain, JSON)
│   └── Main.purs           -- Entry point
```

```
│
└── Git/
    ├── Integration.purs    -- Auto-commit, hooks
    └── Sync.purs           -- (Phase 2) Pull/push coordination
```

Phase 1 Implementation Plan

Milestone 1: Types + JSONL (foundation)

1. Define core types in `Beads.Core.Types`
2. Implement JSON codecs (encode/decode)
3. Implement JSONL read/write
4. Write roundtrip tests

Deliverable: Can read/write issues.jsonl from Go version.

Milestone 2: Graph + Ready (the key feature)

1. Build `IssueGraph` from `Array Issue`
2. Implement `blockedBy`, `isOpen`, `hasNoOpenBlockers`
3. Implement `ready` algorithm
4. Test against known graphs

Deliverable: `ready` function matches Go version output.

Milestone 3: CLI (usable tool)

1. Set up CLI parser (optparse or similar)
2. Implement `init`, `create`, `list`, `show`
3. Implement `update`, `close`, `reopen`
4. Implement `dep add`, `dep remove`
5. Implement `ready`
6. Add `--json` output mode

Deliverable: Usable CLI that can manage issues.

Milestone 4: ID Generation (compatibility)

1. Implement hash-based ID generation
2. Implement adaptive length (4→5→6 chars)
3. Test collision probability
4. Ensure IDs are compatible with Go version

Deliverable: Can create issues that won't collide with Go-created issues.

Decisions

1. **Node.js backend** — Most mature PureScript CLI tooling, FFI for fs/path/process well-established.

Open Questions

1. **SQLite timing**: Skip entirely for Phase 1? The Go version uses it for performance, but we might not need it initially.
2. **Git integration**: Auto-commit after changes? The Go version has hooks and daemon. We could start simpler.
3. **Compatibility goal**: Aim for full compatibility with Go version's JSONL? Or just "inspired by"?
4. **Visualization**: Add PSD3 force-directed graph of dependencies? Would be a nice showcase.
5. **Chemistry phase**: How important is proto/mol/wisp? Could skip for MVP.
6. **Project location**: New repo? Under `tools/`? Standalone showcase?

Potential Extensions

Dependency Visualization

```
-- Use psd3-simulation to visualize the dependency graph
visualizeDeps :: IssueGraph -> Effect Unit
visualizeDeps graph = do
  let nodes = Map.keys graph # Array.fromFoldable
  let edges = graph
        # Map.toUnfoldable
        # concatMap (\(Tuple id issue) ->
            issue.dependencies # map (\d -> { source: id, target: d.toId,
type: d.depType }))

  -- Render with force simulation
  renderForceGraph { nodes, edges }
```

Hylograph Integration

The dependency graph is a tree (well, DAG) — could be a HATS demo:

- Issues as nodes
- Dependencies as edges
- Status as color
- Priority as size
- `ready` issues highlighted

Related Documents

- Original beads: https://github.com/steveyegge/beads
- `docs/ARCHITECTURE.md` in beads repo
- `CHEMISTRY_PATTERNS.md` for proto/mol/wisp semantics