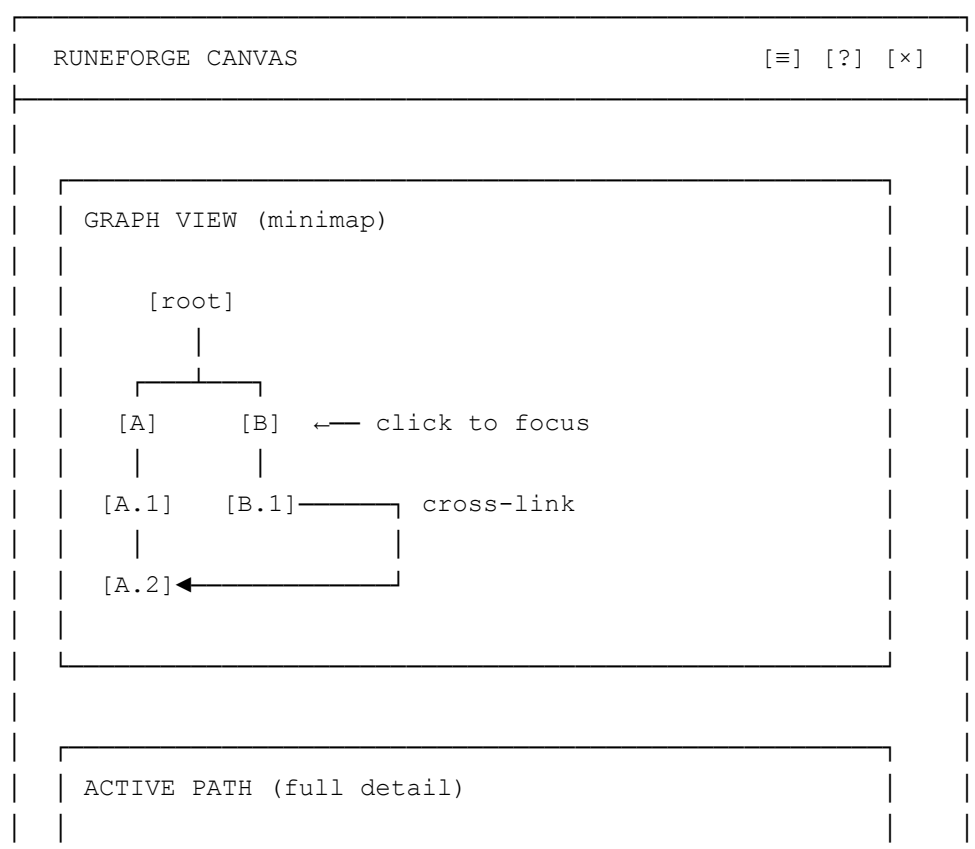
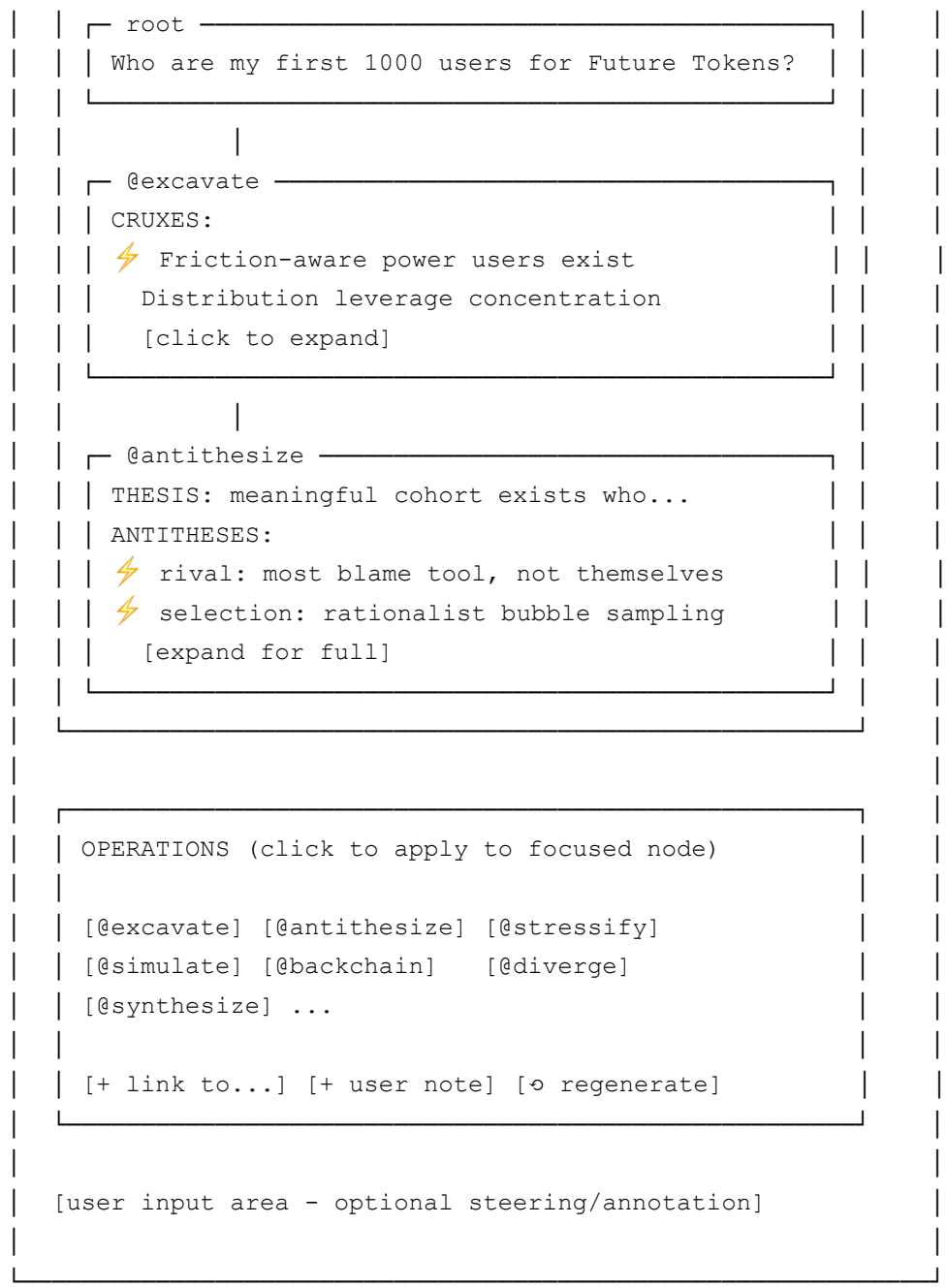
```
"""
Runeforge Canvas - Textual Prototype Sketch

Key insight: Research contexts are ENTANGLED, not separable.
Unlike engineering (parallel workers → merge), research threads
cross-pollinate. Insight in branch A changes what you'd ask in branch B.

Architecture difference from Claude Chic:
- Chic: linear conversation with expandable tool calls (gets slow because ALL his
- Canvas: graph of nodes, only ACTIVE PATH renders fully, siblings show compresse

This sketch shows the core data structures and widget hierarchy.
"""

from __future__ import annotations
from dataclasses import dataclass, field
from typing import Optional, Callable
from datetime import datetime
from enum import Enum
import asyncio

# ============================================================
# CORE DATA MODEL
# ============================================================

class NodeType(Enum):
    ROOT = "root"           # starting question/context
    OPERATION = "operation" # result of a runeforge op
    USER = "user"           # user annotation/steering
    LINK = "link"           # cross-reference to another node


@dataclass
class CanvasNode:
    """Single node in the thinking graph."""
    id: str
    type: NodeType
    operation: Optional[str]  # e.g., "@antithesize", "@excavate"
    content_compressed: str   # always available, ≤100 chars
    content_full: str         # expanded view
    parent_id: Optional[str]
    children_ids: list[str] = field(default_factory=list)
    links_to: list[str] = field(default_factory=list)  # cross-references
    created_at: datetime = field(default_factory=datetime.now)
```

```
    is_active: bool = False   # part of current focus path

    # for research entanglement: when this node was created,
    # which other nodes were "in context"? allows replay/recompute
    context_snapshot: list[str] = field(default_factory=list)


@dataclass
class Canvas:
    """The full thinking graph."""
    nodes: dict[str, CanvasNode] = field(default_factory=dict)
    root_id: Optional[str] = None
    active_path: list[str] = field(default_factory=list)  # ids from root to curr

    def add_node(self, node: CanvasNode) -> None:
        self.nodes[node.id] = node
        if node.parent_id and node.parent_id in self.nodes:
            self.nodes[node.parent_id].children_ids.append(node.id)

    def set_focus(self, node_id: str) -> None:
        """Recompute active path from root to this node."""
        path = []
        current = node_id
        while current:
            path.append(current)
            current = self.nodes[current].parent_id
        self.active_path = list(reversed(path))

        # mark active/inactive
        for nid, node in self.nodes.items():
            node.is_active = nid in self.active_path

    def get_context_for_operation(self, node_id: str) -> list[CanvasNode]:
        """
        Key research insight: context isn't just the parent chain.
        It's the parent chain PLUS any linked nodes PLUS siblings
        that were created before this node.

        This is where research differs from engineering.
        """
        context = []

        # 1. full parent chain
        for pid in self.active_path:
            if pid == node_id:
                break
            context.append(self.nodes[pid])
```

```
        # 2. explicit cross-links
        node = self.nodes[node_id]
        for link_id in node.links_to:
            if link_id in self.nodes:
                context.append(self.nodes[link_id])

        # 3. siblings created before this node (entanglement)
        if node.parent_id:
            parent = self.nodes[node.parent_id]
            for sib_id in parent.children_ids:
                sib = self.nodes[sib_id]
                if sib.created_at < node.created_at and sib_id != node_id:
                    # include compressed form of sibling work
                    context.append(sib)

        return context


# ============================================================
# TEXTUAL WIDGETS (sketch)
# ============================================================

"""
Layout concept:
```





Key UX decisions:

1. GRAPH VIEW is a minimap, not the main interaction surface
   - shows structure at a glance
   - click to change focus
   - doesn't try to render full content (that's what kills chic)

2. ACTIVE PATH renders ONLY the focused branch in full
   - parent chain from root to current focus
   - siblings show as compressed pills, not full content
   - this solves the "gets slow" problem

3. OPERATIONS panel is always visible

```
   - one click to apply
   - no copy-paste cycle
   - result spawns as child of focused node

4. CROSS-LINKS are first-class
   - can link any node to any other
   - links visible in minimap
   - linked content pulled into context for operations

5. CONTEXT MANAGEMENT is explicit
   - when you run an operation, you see what context it's getting
   - can manually add/remove context nodes
   - supports research entanglement without O(n) render cost
"""


# ============================================================
# TEXTUAL APP STRUCTURE (pseudocode with real Textual patterns)
# ============================================================

TEXTUAL_APP_SKETCH = """
from textual.app import App, ComposeResult
from textual.containers import Container, Horizontal, Vertical, ScrollableContain
from textual.widgets import Static, Button, Input, Tree
from textual.reactive import reactive


class NodeWidget(Static):
    '''Single node in the active path view.'''

    expanded = reactive(False)

    def __init__(self, node: CanvasNode) -> None:
        super().__init__()
        self.node = node

    def compose(self) -> ComposeResult:
        # header with operation name + expand toggle
        yield Static(f"— {self.node.operation or 'root'} —", classes="node-header

        if self.expanded:
            yield Static(self.node.content_full, classes="node-content-full")
        else:
            yield Static(self.node.content_compressed, classes="node-content-comp

    def on_click(self) -> None:
        self.expanded = not self.expanded
```

```
class GraphMinimap(Static):
    '''Minimap showing graph structure. Click to navigate.'''

    def __init__(self, canvas: Canvas) -> None:
        super().__init__()
        self.canvas = canvas

    def render_graph(self) -> str:
        # simple ASCII graph rendering
        # in production: use rich.tree or custom box-drawing
        lines = []
        # ... recursive tree render ...
        return "\\n".join(lines)

    def on_click(self, event) -> None:
        # determine which node was clicked based on position
        # emit message to change focus
        pass


class OperationsPanel(Container):
    '''Grid of operation buttons.'''

    OPERATIONS = [
        "@excavate", "@antithesize", "@stressify",
        "@simulate", "@backchain", "@diverge",
        "@synthesize", "@perspectivize", "@metaphorize",
    ]

    def compose(self) -> ComposeResult:
        with Horizontal():
            for op in self.OPERATIONS:
                yield Button(op, id=f"op-{op[1:]}", classes="op-button")

    def on_button_pressed(self, event: Button.Pressed) -> None:
        op_name = event.button.label
        self.post_message(self.RunOperation(op_name))

    class RunOperation(Message):
        def __init__(self, operation: str) -> None:
            self.operation = operation
            super().__init__()


class ActivePathView(ScrollableContainer):
```

```
    '''Renders the focused branch with full detail.'''

    def __init__(self, canvas: Canvas) -> None:
        super().__init__()
        self.canvas = canvas

    def compose(self) -> ComposeResult:
        for node_id in self.canvas.active_path:
            node = self.canvas.nodes[node_id]
            yield NodeWidget(node)

    def refresh_path(self) -> None:
        # clear and recompose when focus changes
        self.remove_children()
        for node_id in self.canvas.active_path:
            node = self.canvas.nodes[node_id]
            self.mount(NodeWidget(node))


class RuneforgeCanvas(App):
    '''Main application.'''

    CSS = '''
    NodeWidget {
        border: solid $primary;
        margin: 1;
        padding: 1;
    }

    .node-header {
        background: $primary-darken-2;
        text-style: bold;
    }

    .op-button {
        min-width: 14;
        margin: 0 1;
    }

    GraphMinimap {
        height: 30%;
        border: solid $secondary;
    }

    ActivePathView {
        height: 50%;
    }
```

```
OperationsPanel {
    height: 20%;
    border: solid $accent;
}
'''

def __init__(self) -> None:
    super().__init__()
    self.canvas = Canvas()

def compose(self) -> ComposeResult:
    yield GraphMinimap(self.canvas)
    yield ActivePathView(self.canvas)
    yield OperationsPanel()

async def on_operations_panel_run_operation(self, message: OperationsPanel.Ru
    # 1. get current focus node
    focus_id = self.canvas.active_path[-1] if self.canvas.active_path else No
    if not focus_id:
        return

    # 2. gather context (this is where research entanglement happens)
    context_nodes = self.canvas.get_context_for_operation(focus_id)

    # 3. build prompt from operation + context
    prompt = self.build_prompt(message.operation, context_nodes)

    # 4. call claude via agent SDK (same as chic)
    result = await self.call_claude(prompt)

    # 5. create new node as child of focus
    new_node = CanvasNode(
        id=generate_id(),
        type=NodeType.OPERATION,
        operation=message.operation,
        content_compressed=result[:100] + "..." if len(result) > 100 else res
        content_full=result,
        parent_id=focus_id,
        context_snapshot=[n.id for n in context_nodes],
    )
    self.canvas.add_node(new_node)

    # 6. auto-focus new node
    self.canvas.set_focus(new_node.id)

    # 7. refresh UI
```

```
        self.query_one(ActivePathView).refresh_path()
        self.query_one(GraphMinimap).refresh()

    def build_prompt(self, operation: str, context: list[CanvasNode]) -> str:
        # load operation definition from runeforge skill
        # inject context nodes
        # return full prompt
        pass

    async def call_claude(self, prompt: str) -> str:
        # use claude-agent-sdk with existing oauth creds
        # same pattern as chic
        pass


if __name__ == "__main__":
    app = RuneforgeCanvas()
    app.run()
"""


# ============================================================
# PERFORMANCE INSIGHT: WHY THIS WON'T GET SLOW
# ============================================================

"""
Chic's problem: it renders EVERY message in the conversation history
as an expandable widget. With long conversations, you have hundreds
of widgets, each with click handlers, each in the DOM equivalent.
Scrolling recalculates layout for all of them.

Canvas solution:

1. MINIMAP is O(n) nodes but each node is just a small box/label,
   no rich content. Constant-time render per node.

2. ACTIVE PATH is O(depth) not O(total_nodes).
   A graph with 500 nodes but max depth 10 only renders 10 widgets.

3. SIBLINGS are compressed pills, not full widgets.
   When you focus [A.2], you see:
   - full content for: root → A → A.2
   - pill for: B (sibling of A)
   - pill for: A.1 (sibling of A.2)

4. LAZY EXPANSION: content_full only loaded when explicitly expanded.
   Most nodes show content_compressed (≤100 chars).
```

```
5. VIRTUALIZATION possible: if active path gets very long,
   can virtualize with textual's scrolling patterns.

This is the key architectural difference that lets you think
in graphs without O(n) render cost.
"""


# ============================================================
# CROSS-LINKING: THE RESEARCH ENTANGLEMENT FEATURE
# ============================================================

"""
Scenario: You're exploring "who are my first 1000 users?"

Path A: @excavate → crux: "friction-aware users exist"
        → @antithesize → "most blame the tool"

Path B: @excavate → crux: "distribution leverage"
        → @simulate → "viral coefficient < 1 unless..."

You realize: the @antithesize output in path A is relevant
to understanding the @simulate output in path B.

With CROSS-LINKS:
1. Focus on B.simulate
2. Click [+ link to...]
3. Select A.antithesize from graph view
4. Now when you run next operation on B.simulate,
   context includes A.antithesize content

This is explicit entanglement management.
You're not forced to either:
- keep everything in one linear thread (chic model, gets slow)
- keep branches fully separate (loses cross-pollination)

Instead: branches are separate by default, cross-linked when useful.
"""


# ============================================================
# AUTH: PIGGYBACK ON CLAUDE CODE
# ============================================================

"""
Same pattern as chic. Read from:
```

```
- ~/.claude/.credentials.json (linux)
- keychain (macos)

Token structure:
{
  "claudeAiOauth": {
    "accessToken": "sk-ant-oat01-...",
    "refreshToken": "sk-ant-ort01-...",
    "expiresAt": 1748658860401,
    "scopes": ["user:inference", "user:profile"]
  }
}

Use claude-agent-sdk to make calls.
Marginal cost = 0 after login (within subscription limits).
"""


if __name__ == "__main__":
    # this file is a sketch, not runnable code
    print("Runeforge Canvas Sketch")
    print("See docstrings and comments for architecture details")
```