Design Decision: Essential vs Optional Information

Problem Statement

The core challenge in prompt refinement is determining what information is required to create a useful, actionable prompt versus what information adds value but is not blocking.

Design Philosophy

I organized information by semantic certainty rather than priority or actors.

Essential Information means what is required to start building.
Optional Information means what helps build it better.

This mirrors real product development. Teams often begin with a vision and core features, while constraints such as budget or timeline are discovered or negotiated later.

Essential Information (Must Have)

1. **Intent Purpose**

Why it is essential:
Without knowing what needs to be built, no meaningful refinement is possible.

Validation rule:

```
'intent.purpose': {
    'min_length': 10,
    'rejection_reason': 'No clear product/system intent'
}
```

Examples:

- Valid: “Build a food delivery mobile app”
- Invalid: “Make something cool”

2. **Problem Being Solved**

Why it is essential:
This defines the reason behind the request. Without understanding the problem, it is impossible to evaluate whether the extracted requirements make sense.

Validation rule:

```
'intent.problem_being_solved': {
    'min_length': 10,
    'rejection_reason': 'Problem statement missing or unclear'
}
```

Examples:

- Valid: “Users struggle to find nearby restaurants quickly”
- Invalid: Empty or unspecified

3. At Least One Requirement

Why it is essential:
A prompt with zero requirements is not actionable, even if intent is clear.

Validation rule:

```
'requirements': {
    'min_count': 1,
    'rejection_reason': 'No actionable requirements identified'
}
```

Examples:

- Valid: [“User can search restaurants”, “User can place orders”]
- Invalid: Empty list

Optional Information (Nice to Have)

1. **Constraints**

Why it is optional:
Many real projects begin without explicit constraints.

Value when present:

- Helps scope the solution
- Prevents over-engineering
- Sets realistic expectations

Examples:

- Budget: $50,000
- Timeline: 6 months
- Platform: Must support iOS 14+

2. Deliverables

Why it is optional:
Deliverables are often implicit in the requirements.

Value when present:

- Makes expectations explicit
- Helps with planning and estimation
- Useful for proposals or contracts

Examples:

- iOS mobile app
- API documentation
- Admin dashboard

3. Assumptions

Why it is optional:
Assumptions emerge during analysis rather than being explicitly provided.

Value when present:

- Documents inferred decisions
- Highlights risks
- Makes reasoning transparent

Examples:

- “Assuming single-city launch initially”
- “Assuming English-only for MVP”

Completeness Scoring

Information is weighted by impact on actionability rather than simple presence.

```
COMPLETENESS_WEIGHTS = {
    'intent': 0.30,
    'requirements': 0.40,
    'constraints': 0.15,
    'deliverables': 0.10,
    'no_conflicts': 0.05
}
```

Rationale

Requirements (40%)
Without clear functionality, nothing can be built. This is the core value of refinement.

Intent (30%)
Clear purpose and problem definition drive all downstream decisions.

Constraints (15%)
Important for feasibility, but often discovered later.

Deliverables (10%)
Often derivable from requirements.

No conflicts (5%)
A bonus for clarity, not a blocker.

Why This Design Beats Alternatives

Alternative 1: Reject if any field is missing
Problem: Too strict. Real-world prompts are often incomplete.
Example: “Build Uber for plumbers” is valid despite missing budget or timeline.

Alternative 2: Accept everything without validation
Problem: Garbage in, garbage out.
Example: “Make an app” would incorrectly pass.

Chosen Approach: Essential validation plus completeness scoring

Advantages:

- Rejects truly unusable inputs
- Accepts imperfect but actionable prompts
- Provides a quality signal instead of binary rejection

Examples:

- “Build Uber for plumbers” → Valid, low completeness (~0.60)
- “Build e-commerce with detailed features” → Valid, high completeness (~0.95)
- “Write a poem” → Invalid (no product intent)

Edge Cases Handled

Case 1: Vague but Valid
Input: “I need a food delivery app”
Decision: Accept with low completeness
Intent present, basic requirement inferred, details missing.

Case 2: Detailed but Non-Product
Input: Long essay about AI ethics
Decision: Reject
No product or system intent.

Case 3: Requirements Without Context
Input: List of 20 features with no explanation
Decision: Accept with medium completeness
Requirements exist, but context is weak.

Case 4: Conflicting Information
Input: “Build an offline app that syncs to the cloud”
Decision: Accept but flag conflict
Intent and requirements clear, contradiction documented, score reduced.

Implementation Details

Validation happens in two stages.

Stage 1: LLM Extraction (refiner.py)
The LLM extracts intent, requirements, constraints, and assumptions using best-effort reasoning.

Stage 2: Explicit Validation (validation.py)
Programmatic checks enforce essential fields and compute completeness scores.

Why this separation exists:

The LLM handles ambiguity, inference, and interpretation.
Code handles rules, consistency, and explainability.

This makes the system auditable and predictable.

Self-Critique

Strengths:

- Clear separation between must-have and nice-to-have
- Flexible enough for real-world inputs
- Strict enough to reject garbage
- Explainable scoring system

Weaknesses:

- Weight values are heuristic
- Minimum length thresholds are arbitrary
- Does not account for domain-specific requirements

Future Improvements:

- Configurable weights per domain
- Feedback-driven tuning
- Custom validation rules for specific industries

Conclusion

Essential versus optional is not about perfection. It is about actionability.

This design ensures that every accepted prompt contains enough information to start meaningful work, while remaining forgiving enough to handle real-world, imperfect inputs. The completeness score provides a quality signal without enforcing rigid pass/fail behavior.