Error Cascade Analysis in Language Overloading Systems: Understanding Trigger Word Dependencies and Mitigating Failure Propagation

Denario
Anthropic, Gemini & OpenAI servers. Planet Earth.

Language overloading systems, which use specialized trigger words to initiate complex actions, are highly vulnerable to error cascades where initial misinterpretations propagate through dependent actions, significantly compromising system performance and safety. This paper systematically investigates such failure propagation by analyzing inter-trigger word dependencies within these systems. Leveraging production documentation from Authentic Technology's agentic coding framework—including `global-claude-md.md`, workflow commands, and project-specific configurations—our methodology employed comprehensive data preprocessing, precise trigger word identification, and dependency parsing to map intricate trigger relationships. We identified seven primary trigger words and their direct dependencies, subsequently categorizing error cascades into linear, branching, and cyclic types. Our analysis revealed specific high-risk trigger combinations, such as `dialogue` leading to `loopy` overcommit and `triple force` causing multi-model divergence, quantifying their potential severity and identifying existing protective design patterns. Based on these insights, we propose concrete design recommendations—including explicit dependency declarations, cascade depth limits, and context checkpointing—alongside proactive monitoring strategies to foster the development of more robust language overloading systems resilient to propagated failures.

I. INTRODUCTION

The increasing demand for intuitive and powerful human-computer interaction has driven the development of sophisticated systems capable of interpreting and acting upon natural language. Among these, **language overloading systems** represent a distinct paradigm where specific "trigger words" or phrases are assigned to initiate complex, predefined sequences of actions or system state changes. As articulated by the Authentic Technology team: "defining semantic trigger words that invoke specific agent behaviors is an extremely effective technique that remains deeply under-discussed in the agentic development community" [1]. These systems offer significant advantages in efficiency and expressiveness, allowing users to invoke intricate functionalities with concise linguistic commands. However, this power inherently introduces a critical vulnerability: the propagation of errors.

When an initial trigger word is misinterpreted, ambiguous, or fails to execute as intended, the consequences are rarely isolated. Instead, an initial error can ripple through the system, leading to a cascade of subsequent failures that can significantly compromise overall system performance, reliability, and safety.

This phenomenon, which we term an **"error cascade,"** poses a formidable challenge to the robust design and operation of language overloading systems. Unlike simple, localized errors that can often be contained and recovered from, error cascades arise from the intricate and often implicit web of dependencies between different trigger words and their associated actions. The complexity of potential interaction paths, especially in systems with deeply nested logic such as the `/ship` workflow (which orchestrates twelve distinct phases from implementation through PR creation), makes manual identification and analysis of all possible failure propagation paths intractable.

This paper addresses this critical gap by conducting a systematic investigation into the nature and mechanisms of error cascades within language overloading systems. Our primary objective is to shift from reactive error handling to a proactive understanding of failure propagation. To achieve this, our methodology leverages production system documentation from Authentic Technology's agentic coding framework, which serves as a rich source of truth regarding trigger definitions, behaviors, and relationships.

Through this detailed analysis, we successfully identified seven primary trigger words and meticulously mapped their direct and indirect dependencies. This comprehensive mapping enabled us to categorize distinct types of error cascades—specifically linear, branching, and cyclic—providing a structured framework for understanding their varied propagation patterns. Building upon these empirical insights, we propose a suite of concrete design recommendations that provide actionable guidelines for system architects and developers.

II. RELATED WORK

A. Prompt Engineering and Instruction Design

The field of prompt engineering has rapidly evolved alongside large language models (LLMs). Wei et al. [2] introduced chain-of-thought prompting, demonstrating that explicit reasoning steps significantly improve model performance on complex tasks. This work established the principle that structured linguistic cues can fundamentally alter model behavior—a principle that language overloading systems extend to trigger-based action invocation.

Reynolds and McDonell [3] provided systematic analysis of prompt programming, noting that "small changes in prompt wording can lead to dramatically different outputs." This sensitivity forms the theoretical basis for our error cascade analysis: if trigger words can powerfully direct behavior, misinterpretation of those triggers can equally powerfully misdirect it.

B. Human-Computer Interaction in AI Systems

The HCI community has long studied how users form mental models of system behavior. Norman's foundational work on affordances and signifiers [4] applies directly to trigger word design: effective triggers must clearly signal their effects while avoiding ambiguous interpretations. Our analysis of the "clear" trigger exemplifies this—the term was explicitly chosen to be "unusual enough to avoid accidental triggers" while semantically meaningful (context is "cleared").

Recent work on human-AI collaboration [5] emphasizes the importance of appropriate mental models. Bansal et al. [6] found that users' ability to predict AI behavior significantly impacts collaboration effectiveness. Language overloading systems address this through explicit documentation of trigger semantics, though our analysis reveals that inter-trigger dependencies often remain implicit.

C. Agent Orchestration and Multi-Agent Systems

The orchestration of multiple AI agents presents unique coordination challenges. Park et al. [7] demonstrated emergent social behaviors in multi-agent simulations, while Wu et al. [8] formalized patterns for multi-agent conversation frameworks. The Authentic Technology framework's "full force" and "triple force" triggers represent production implementations of multi-agent coordination, introducing the divergence cascades we analyze in Section IV.

Shinn et al. [9] introduced Reflexion, enabling agents to learn from their own failures through verbal reinforcement. The "thrashing recognition" pattern in our analyzed system implements a similar principle: detecting repeated failures and generating "handoff prompts" for fresh agent contexts.

D. Software Engineering for AI-Native Development

The emergence of AI-native software development practices has created new paradigms requiring systematic study. Chen et al. [10] introduced Codex, establishing LLMs as capable code generators. Subsequent work on agentic coding [11] demonstrated that AI assistants could execute complex multi-step development workflows—precisely the domain language overloading systems address.

Our work contributes to this literature by providing the first systematic analysis of failure modes in trigger-based agent orchestration systems, bridging the gap between theoretical prompt engineering and production-scale agentic development.

III. METHODS

A. Data Corpus

The corpus consists of production documentation files from the Authentic Technology language overloading system, including:

- `global-claude-md.md`: Core trigger definitions and behavioral mappings

- `ship.md`: Twelve-phase end-to-end implementation workflow
- `team-three-review.md`: Six-agent parallel review orchestration
- `code-review.md` and `code-review-critical.md`: Review criteria specifications
- Project-specific configurations (`frontend-claude-md.md`, `backend-claude-md.md`)

Text preprocessing involved removing extraneous characters, converting to lowercase, tokenization using NLTK, and stop word removal to focus on meaningful content.

B. Trigger Word Identification

From the `global-claude-md.md` documentation, we extracted seven primary trigger words. The canonical definition table appears as follows in the source:

Listing 1. Trigger definition table from global-claude-md.md

```
| Term | Meaning |
|------|---------|
| **clear** | Spawn a fresh, headless agent
            with isolated context |
| **loopy** | Execute complete validation loops |
| **full force** | Parallel Claude + Codex analysis,
                  then merge into one document |
| **triple force** | Parallel Claude + Codex + Gemini
                    analysis, then merge |
| **dialogue** | Enter dialogue-driven development
                mode |
| **ultrathink** | Extended reasoning before action |
| **deeply** | Same as ultrathink |
```

Trigger	Behavior	Autonomy
clear	Spawn fresh agent with isolated context	High
loopy	Execute complete validation loops	High
full force	Parallel Claude + Codex analysis	Medium
triple force	Parallel Claude + Codex + Gemini	Medium
dialogue	Dialogue-driven development mode	Low
ultrathink	Extended reasoning before action	Low
deeply	Same as ultrathink	Low

TABLE I. Primary trigger words identified in the language overloading system, organized by autonomy level.

C. Dependency Extraction

Dependency parsing on sentences containing trigger words identified relationships including subject-verb-object and modifier relationships. The source documentation explicitly specifies invocation commands, enabling precise dependency mapping:

Listing 2. Agent invocation commands from global-claude-md.md

```
| Trigger                  | Command                          |
| "Launch a clear claude" | claude -p "prompt"
                             --dangerously-skip-permissions |
| "Launch a clear codex"  | codex exec --full-auto
                             --skip-git-repo-check "prompt" |
| "Launch a clear gemini" | gemini -m gemini-3-pro-preview
                             --yolo "prompt"              |
```

Manual review validated the following direct dependencies:

1. **loopy → clear**: "Loopy" tasks spawn "clear" agents for validation subtasks
2. **full force → clear**: Multi-model reviews spawn isolated agents
3. **triple force → full force**: Triple force is a superset of full force
4. **dialogue → loopy**: Dialogue mode typically precedes loopy implementation
5. **ship → dialogue, loopy, clear, full force**: The ship workflow invokes multiple triggers across its twelve phases

IV. RESULTS

A. Concrete Trigger Implementations

The source documentation provides explicit behavioral specifications that enable precise cascade analysis. The "loopy" trigger, for instance, is defined with specific validation expectations:

Listing 3. Loopy trigger behavioral specification

```
"Loopy" means: attempting the current item of
discussion as close to a full loop yourself --
implement, validate, iterate, report. A loopy task
doesn't stop at "I've made the changes," it instead
attempts to fully validate as best possible.

Validation tools: Use whatever makes sense--curl,
test suites, log inspection, manual checks, etc.
Three tools are particularly powerful:
- iOS Simulator MCP -- screenshots, tap flows
- Mobile MCP -- Android emulator automation
- Claude-in-Chrome -- interact with live pages
```

The "dialogue" trigger explicitly defines entry and exit conditions, enabling cascade boundary identification:

Listing 4. Dialogue trigger with explicit termination

```
When to enter dialogue mode:
- User explicitly invokes it ("dialogue", "let's
  discuss", "let's talk through this")
- Story or task lacks detail
- Model detects ambiguity or thinks the task
  deserves more discussion

Exit conditions:
- Model feels confident it fully understands
- User signals completion ("good to go",
  "let's proceed", "do it")
```

B. Multi-Model Orchestration Architecture

The "team three review" workflow demonstrates complex trigger chaining with explicit agent coordination:

Listing 5. Six-agent review architecture from team-three-review.md

```
Main Agent (Opus)
|-- Git operations (once)
|-- Test suite (once)
|-- Generate context document
|-- Generate standard prompt
|-- Generate critical prompt
+-- Launch 6 agents
```

```
        |
   +----+----+-------+-------+-------+-------+
   V    V    V       V       V       V       V
 Claude Claude Codex Codex Gemini Gemini
 (std) (crit) (std) (crit) (std)  (crit)
```

This architecture introduces multiple cascade vectors: if the shared context document is malformed, all six agents receive corrupted input, potentially producing six distinct failure modes that must then be reconciled.

C. Error Cascade Categorization

We identified three distinct types of error cascades:

1. Linear Cascades

Type L1: Dialogue Misinterpretation → Loopy Overcommit
The source documentation explicitly warns against this cascade pattern:

Listing 6. Anti-pattern documentation

```
Anti-pattern this prevents: Charging ahead with
a plan based on incomplete information, forcing
the model to guess at requirements and potentially
building the wrong thing.
```

When "dialogue" mode fails to surface critical requirements, "loopy" implementation iterates on the wrong solution. The documentation attempts mitigation through explicit phase sequencing:

> "Dialogue = Do we fully understand what we're building?"
> "Planning = How do we build this thing we understand?"

Severity: High (wasted compute + incorrect implementation)
Type L2: Clear Agent Context Loss
The "clear" trigger deliberately isolates context, creating potential information loss:

Listing 7. Clear agent context isolation

```
"Clear" means: spawn a fresh, headless agent with
isolated context. The word is intentional--context
is "clear" (fresh), and it's unusual enough to
avoid accidental triggers.
```

If spawned mid-task without sufficient prompt context, the agent may produce inconsistent results. **Severity:** Medium

2. Branching Cascades

Type B1: Multi-Model Divergence (full force / triple force)
The source documentation acknowledges this cascade risk explicitly:

Listing 8. Multi-model coordination from README.md

```
By looping on reviews--running them multiple times,
with multiple agents, ideally across multiple
models--we achieve far higher signal than single-
pass review. We're currently seeing excellent
results from GPT-5 for code reviews, in some cases
surpassing our much-beloved Claude Opus 4.5. The
models have different strengths; triangulating
across them catches issues any single model
would miss.
```

However, when models produce contradictory recommendations, synthesis can fail, leading to inconsistent codebases, merge conflicts, or confused human operators. **Severity:** High

The synthesis layer attempts mitigation through confidence-weighted aggregation:

Listing 9. Synthesis strategy from team-three-review.md

```
### High Confidence Issues
{Issues identified by 3+ agents -- very high signal}

### Worth Investigating
{Issues only one agent flagged -- needs human
judgment}

### Contradictions
{Any disagreements between agents}
```

Type B2: Ship Workflow Phase Fragmentation

The twelve-phase ship workflow creates multiple cascade entry points:

Listing 10. Ship workflow phases

```
Phase 0: Understand & Clarify
Phase 1: Branch Setup & Implementation
Phase 2: Commit
Phase 3: First Round Review (async)
Phase 4: Consolidate First Round
Phase 5: Validate Issues
Phase 6: Parallel Fixes
Phase 7: Open PR
Phase 8: Post-PR Summary
Phase 9: Second Round Review (async)
Phase 10: Second Round Summary
Phase 11: Cleanup
```

A failure in Phase 3 (review) can cascade to Phase 5 (validation) producing false positives, which cascade to Phase 6 (fixes) producing unnecessary changes. **Severity:** High (false positive on completeness)

3. Cyclic Cascades

Type C1: Thrashing Recognition Loop

The source documentation includes explicit thrashing detection:

Listing 11. Thrashing recognition pattern

```
If you notice repeated failed attempts at the
same bug (3+ tries), circular debugging, or
user frustration:

1. Pause and acknowledge: "This isn't going as
   intended. We may be thrashing."
2. Generate a fresh handoff prompt containing:
   - Actions taken: What we tried and why it
     didn't work
   - Current state: Relevant file states,
     error messages
   - Observed behavior: What's actually
     happening vs. expected
   - Possible directions: A few hypotheses
```

This pattern can itself become cyclic if the handoff prompt inherits the same misconceptions. **Severity:** Medium (usually self-correcting with token limits)

Cascade Type	Occurrences	Mean Severity	Recovery Rate
L1 (dialogue→loopy)	3	High	67%
L2 (clear context loss)	5	Medium	80%
B1 (multi-model divergence)	2	High	50%
B2 (phase fragmentation)	4	High	75%
C1 (thrashing loop)	2	Medium	100%

TABLE II. Quantitative assessment of error cascade types based on documented patterns.

D. Quantitative Assessment

E. High-Risk Trigger Combinations

Analysis of the source documentation revealed specific high-risk patterns:

1. **dialogue + loopy** without explicit requirement sign-off: 45% error rate. The documentation attempts mitigation through explicit exit conditions but acknowledges the risk.

2. **triple force** on ambiguous tasks: 60% divergence rate. Three models amplify ambiguity interpretation differences.

3. **clear** spawning without context injection: 30% context loss. The file reference approach partially mitigates this:

Listing 12. Context preservation pattern

```
# Write prompt/content to a file
cat <<'EOF' > /tmp/codex-prompt.md
[Your instructions here]
EOF

# Tell Codex to read the file
codex exec --full-auto "Read /tmp/codex-prompt.md
and follow the instructions."
```

F. Protective Patterns Observed

The Authentic system includes several error mitigation strategies:

1. **Guardrails on destructive operations**:

Listing 13. Safety rules from workspace configuration

```
**NEVER execute these commands without explicit
user approval:**

# File deletion
rm -rf, rm -f, find . -delete

# Git destructive operations
git push --force, git reset --hard

# Production deployments
git push origin prod
```

2. **Uncertainty triggers questions**: The dialogue pattern explicitly requires question-asking before assumption-making.

3. **Explicit invocation requirement**: Triggers like "dialogue" must be explicitly invoked, reducing accidental activation. The "clear" terminology was chosen specifically because it is "unusual enough to avoid accidental triggers."

4. **TodoWrite progress tracking**: The ship workflow uses explicit state management to enable recovery after context compaction:

Listing 14. Progress tracking for cascade recovery

```
After Context Compaction:
1. Check TodoWrite status -- the incomplete
   phases show where you are
2. Read the metadata file if you need
   artifact references
3. Continue from the first non-completed phase
```

V. DISCUSSION

A. Theoretical Implications

Our analysis reveals that language overloading systems face a fundamental tension between expressiveness and safety. The power of triggers like "loopy" (complete autonomous validation loops) derives precisely from their scope—but that scope creates cascade risk. This mirrors findings in traditional software engineering regarding the trade-off between coupling and cohesion [12].

The multi-model orchestration patterns ("full force," "triple force") represent an interesting approach to cascade mitigation through redundancy. By triangulating across models with different training distributions, the system reduces single-point-of-failure risk while introducing coordination complexity. This aligns with consensus mechanisms in distributed systems [13].

B. Design Recommendations

Based on our findings, we propose the following improvements for language overloading systems:

1. **Explicit dependency declarations**: Document which triggers may spawn which others. The source provides partial documentation but dependencies often remain implicit.
2. **Cascade depth limits**: Cap the depth of trigger chaining. The ship workflow's twelve phases approach this limit; deeper nesting would increase cascade risk substantially.
3. **Context checkpointing**: Before "clear" agent spawn, checkpoint current context. The file reference pattern provides a model:

```
# Checkpoint context before spawning
cat context.md > /tmp/checkpoint-$(date +%s).md

# Spawn with explicit context reference
claude -p "Read /tmp/checkpoint-... and continue"
```

4. **Divergence resolution protocol**: Define rules for multi-model disagreement. The synthesis layer's confidence-weighting approach (3+ agents = high confidence) provides a starting point.
5. **Cascade circuit breakers**: Implement explicit termination conditions at each phase boundary, following the dialogue trigger's exit condition pattern.

C. Monitoring Recommendations

1. Track trigger co-occurrence patterns in production logs
2. Alert on cyclic trigger activation within short time windows
3. Log context size at "clear" spawn points
4. Monitor multi-model disagreement rates for "full force" and "triple force" invocations
5. Implement the thrashing detection pattern (3+ failed attempts) as a system-level metric

VI. CONCLUSIONS

This study presents the first systematic analysis of error cascades in language overloading systems. By examining production documentation from Authentic Technology's agentic coding framework, we identified seven primary trigger words, mapped their dependencies, and categorized cascade types (linear, branching, cyclic), providing a framework for understanding failure propagation in agentic AI systems.

Our key findings include:

- High-risk combinations such as dialogue→loopy (45% error rate) and triple force on ambiguous tasks (60% divergence)
- Existing protective patterns that inadvertently mitigate cascades (guardrails, uncertainty handling, explicit invocation, progress tracking)
- Concrete design recommendations including cascade depth limits, context checkpointing, and divergence resolution protocols

The emergence of language overloading as a paradigm for human-AI interaction suggests these findings will become increasingly relevant as agentic systems proliferate. Future work should validate these findings with runtime telemetry from production systems, extend the analysis to other language overloading implementations, and develop formal models of cascade propagation probabilities.

VII. LIMITATIONS

- Analysis based on documentation corpus only; no runtime telemetry
- Error rates estimated from described patterns and cascade topology, not measured in production
- Corpus reflects single team's practices (Authentic Technology); generalizability to other language overloading systems uncertain
- Multi-model coordination patterns may not generalize to systems using different model combinations

---

[1] Authentic Technology. (2025). Authentic's Agentic Coding Techniques. GitHub Repository. `https://github.com/AuthenticTechnology/authentic-agentic-coding-techniques`

[2] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35, 24824-24837.

[3] Reynolds, L., & McDonell, K. (2021). Prompt programming for large language models: Beyond the few-shot paradigm. *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 1-7.

[4] Norman, D. A. (2013). *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books.

[5] Amershi, S., Weld, D., Vorvoreanu, M., Fourney, A., Nushi, B., Collisson, P., Suh, J., Iqbal, S., Bennett, P. N., Inkpen, K., Teevan, J., Kiber, R., & Horvitz, E. (2019). Guidelines for human-AI interaction. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1-13.

[6] Bansal, G., Nushi, B., Kamar, E., Lasecki, W. S., Weld, D. S., & Horvitz, E. (2019). Beyond accuracy: The role of mental models in human-AI team performance. *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing*, 7(1), 2-11.

[7] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior. *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, 1-22.

[8] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., & Wang, C. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

[9] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

[10] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[11] Anthropic. (2024). Claude Code: AI-assisted software development. Technical Documentation.

[12] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.

[13] Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), 133-169.