TrellisPoint
Growth the Strategic Way

Copilot Studio Reference Sheet

Purpose

This reference provides a practical view of how Copilot Studio fits into Park Place's broader agentic architecture. It is intended to help builders design agents that are reliable, maintainable, and easy to extend over time.

Copilot Studio works best as an orchestration layer. It coordinates conversations, triggers actions, and routes work, but it should not be treated as a rules engine or a place to embed complex business logic.

Copilot Studio's Role in the Architecture

Copilot Studio is responsible for:

- Conversational flow and intent handling
- Triggering downstream actions
- Routing work across systems
- Managing lightweight conversational context

Copilot Studio is **not** responsible for:

- Enforcing complex business rules
- Securing secrets or credentials
- Executing long-running or stateful logic
- Acting as a system of record

When logic becomes complex or security-sensitive, responsibility should move downstream.

Core Building Blocks

- Topics – Entry points and flow control for agent behavior
- Entities – Structured data extracted from conversations
- Actions – Calls to Power Automate, Azure Functions, or APIs
- Memory – Short-term context for continuity, not long-term storage
- Triggers – Events from Dynamics, Dataverse, email, queues, or workflows

---

Common Architecture Patterns

- Inbox triage with human approval
- CRM-triggered follow-up agents
- Multi-step workflows coordinated across Power Automate and Azure Functions
- Multi-agent handoffs where Copilot coordinates and downstream services execute

---

Common Failure Patterns

- Excessive logic embedded in prompts
- Missing or poorly defined triggers
- Overuse of memory for data storage
- No escalation or approval path
- Limited logging and traceability

Design Guidelines

- Keep topics small and composable
- Validate inputs early
- Fail visibly and log decisions
- Design escalation before automation
- Treat Copilot Studio as a coordinator, not an executor

Prompt Engineering for Devs Guide

Purpose

This guide provides practical guidance for designing prompts that behave predictably, scale safely, and remain understandable as agentic systems evolve.

Prompts should be treated as system components with clear intent and boundaries, not as creative experiments.

Prompts as System Interfaces

A prompt defines:

- The role the agent is playing
- The scope of responsibility
- The expected output format
- The conditions for escalation or failure

Ambiguity at the prompt level propagates downstream.

Core Design Principles

- Define roles and goals explicitly
- Constrain scope and expected outputs
- Separate reasoning from execution
- Design for failure, not perfection
- Prefer deterministic outputs where possible

Techniques That Matter in Production

- Prompt chaining to reduce complexity
- Context injection over excessive memory use
- Structured outputs (JSON, tables)
- Confidence scoring or uncertainty flags

Managing Hallucinations

- Require assumptions to be stated
- Ask for sources or reasoning summaries
- Cross-check with tools or APIs
- Escalate when confidence thresholds are not met

Token and Cost Awareness

- Remove redundant context
- Reference data instead of copying it
- Split large prompts into stages
- Design for predictable response length

Developer Guidance

Reliable prompts reduce operational risk, improve trust, and control cost. Prompt quality directly affects system stability.

Azure Function Snippets

Purpose

This reference outlines common patterns for extending agents with server-side logic using Azure Functions. It is intended to standardize how execution, security, and integration are handled across agentic solutions.

When to Use Azure Functions

Azure Functions are appropriate when:

- Business logic is complex or conditional
- Secure access to systems is required
- Data enrichment or validation is needed
- Workflows are long-running or asynchronous
- Auditability and observability are required

Logic that affects data integrity or compliance should not live in prompts.

Common Function Patterns

- **HTTP-triggered functions** for agent calls
- **Queue-triggered functions** for asynchronous processing
- **Durable Functions** for multi-step orchestration

Security and Governance Defaults

- Use managed identities
- Store secrets in Azure Key Vault
- Validate inputs rigorously
- Redact sensitive data before LLM interaction

Operational Considerations

- Monitor latency and failures
- Implement retries and timeouts
- Log inputs and outputs responsibly
- Control execution costs

Developer Guidance

Functions provide agents with controlled execution, security, and enterprise-grade reliability. They are a key part of scaling agentic systems responsibly.